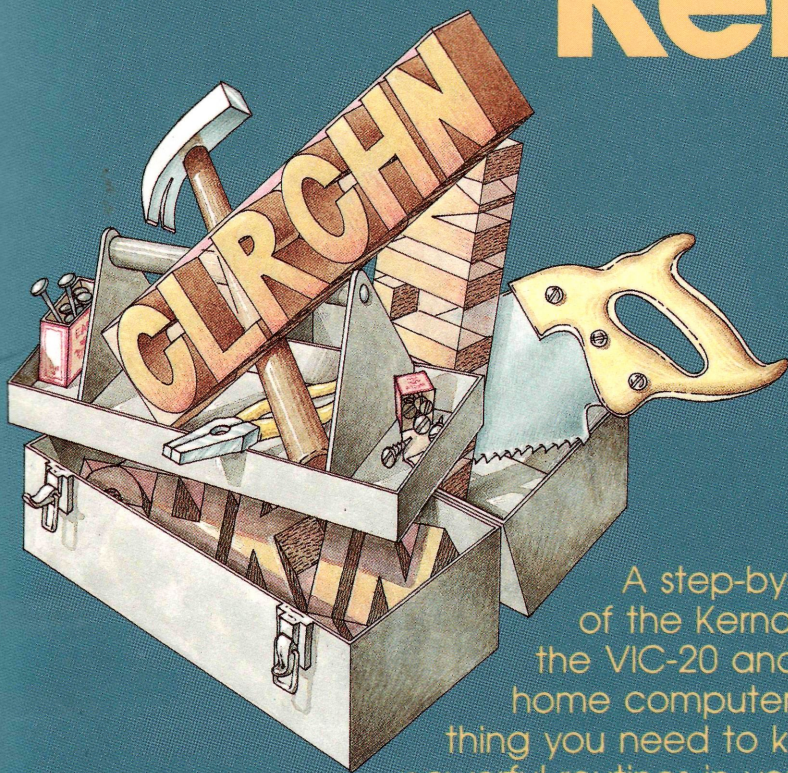


**COMPUTE!'s**  
VIC-20 and Commodore 64

# Tool Kit: Kernal



Dan Heeb

A step-by-step explanation of the Kernal ROM routines of the VIC-20 and Commodore 64 home computers. Includes everything you need to know to use these powerful routines in your own programs.



**COMPUTE!'s**  
VIC-20 and Commodore 64  
**Tool Kit:**  
**Kernal**

Dan Heeb

Copyright 1985, COMPUTE! Publications, Inc. All rights reserved

Reproduction or translation of any part of this work beyond that permitted by Sections 107 and 108 of the United States Copyright Act without the permission of the copyright owner is unlawful.

The author and publisher have made every effort in the preparation of this book to insure the accuracy of the programs and information. However, the information and programs in this book are sold without warranty, either express or implied. Neither the author nor COMPUTE! Publications, Inc., will be liable for any damages caused or alleged to be caused directly, indirectly, incidentally, or consequentially by the programs or information in this book.

Printed in the United States of America

10 9 8 7 6 5 4 3 2

ISBN 0-942386-33-7

COMPUTE! Publications, Inc., Post Office Box 5406, Greensboro, NC 27403, (919) 275-9809, is one of the ABC Publishing Companies, and is not associated with any manufacturer of personal computers. Commodore 64 and VIC-20 are trademarks of Commodore Electronics Limited. COMPUTE! Publications assumes no liability for errors in this book.

# Contents

---

Foreword .....	v
Preface .....	vii
<b>Chapter 1.</b> Interrupts and System Reset .....	1
<b>Chapter 2.</b> System Reset .....	15
<b>Chapter 3.</b> NMI Interrupts .....	31
<b>Chapter 4.</b> IRQ Interrupts .....	43
<b>Chapter 5.</b> Kernal Routines .....	63
<b>Chapter 6.</b> Miscellaneous Routines .....	111
<b>Chapter 7.</b> Screen Routines .....	121
<b>Chapter 8.</b> Serial I/O Routines .....	181
<b>Chapter 9.</b> RS-232 I/O Routines .....	225
<b>Chapter 10.</b> Tape I/O Routines .....	269
<b>Appendices</b>	
<b>A:</b> Commodore 64 and VIC I/O and Video Control Registers .....	399
<b>B:</b> Index of Kernal Routines by Address .....	407
<b>C:</b> Cross Reference of Kernal Routines by Chapter ..	421





# Foreword

---

*COMPUTE!'s VIC-20 and Commodore 64 Tool Kit: The Kernal* is the first detailed description of the built-in programs that run the 64 and VIC. It's everything you need to know to understand and use the operating system of your computer.

*COMPUTE!'s VIC-20 and Commodore 64 Tool Kit: The Kernal* explains routines in groups, so that related routines are usually discussed within the same chapter. By grouping routines together, the book enables you to learn more than just what each section of code does—you'll see how several routines work together.

If you know the 6502/6510 machine language instructions, but programming with them is still a mystery to you, *Tool Kit: The Kernal* can help. Instead of simply listing cryptic assembly language code, it describes what is happening in each routine in easy-to-understand English. You can learn machine language tricks just by looking at the way the professional programmers designed the ROMs in the VIC and 64. And, as with all *COMPUTE!* books, the explanations are clear and concise.

If you are a machine language programming expert, *Tool Kit: The Kernal* is a valuable source book. You'll be able to quickly find the routine you are interested in and follow the logic of the code. The author doesn't waste your time with unnecessary details, and his no-nonsense approach allows you to get the information you want as quickly as possible.

*Tool Kit: The Kernal* is the sequel to *COMPUTE!'s VIC-20 and Commodore 64 Tool Kit: BASIC* (which documents the Commodore BASIC ROM routines), and is an excellent companion to other *COMPUTE!* books (like *Mapping the VIC*, *Mapping the Commodore 64*, *Programming the VIC*, and *Programming the Commodore 64*). Armed with these guides and a ROM listing, you'll discover all the secrets of your VIC or 64.





# Preface

---

The subject of this book is the part of the Commodore 64 and VIC-20 called the *Kernal*. Some people think of the Kernal as only the standard jump vector table, which is correctly called the *Kernal Jump Table*. But the Kernal is actually larger. It is the section of ROM that handles the input/output (I/O) and system management routines such as the interrupt handlers. The remaining part of ROM that makes up the BASIC language is covered in *COMPUTE!'s VIC-20 and Commodore 64 Tool Kit: BASIC*.

The 6502 chip is the microprocessor in the VIC-20, while the Commodore 64 has the 6510. Both chips use the same machine language (ML) instruction set. While it is not absolutely essential that you know 6510/6502 machine language to read this book, it would help. Numerous books on 6510/6502 machine language are available, so this book does not attempt to teach ML programming.

Due to copyright restrictions, a commented disassembly of the code for the Kernal is not presented in this book. Using any machine language monitor, you should be able to view the disassembled Kernal on your screen and follow along with the comments about the code that are found here. At least two books are available that contain printed listings of the Kernal instructions, if you find it easier to read rather than view them on the screen. These are *The Anatomy of the Commodore 64* (from Abacus Software) and *Inside the Commodore 64* (by Milton Bathurst).

The main purpose of the Kernal is to allow communication with the various I/O devices—the screen, keyboard, and cassette drive, as well as RS-232-C devices, disk drives, and printers. The interrupt routines are crucial in the functioning of I/O handling, as interrupts allow an I/O device to notify the 6510/6502 that it needs servicing.

The information in this book applies to both the Commodore 64 and the VIC-20. When an address differs, the addresses are specified with a slash between two numbers; the number on the left gives the Commodore 64 address and the right gives the VIC-20. For example, EA31/EABF refers to location EA31 *hexadecimal* on the 64 and EABF *hex* on the

VIC-20. Whenever a hex number represents an address, it is written simply as the number (say, EA31), while an absolute hex number used by the accumulator or one of the registers is preceded by a dollar sign (for example, \$25). When an address appears in parentheses, it refers to a two-byte pointer, or *vector*, consisting of the address in parentheses and the following location. For example, (FFFC) means the two-byte vector consisting of locations FFFC and FFFD. Decimal numbers are usually indicated by the word *decimal* following in parentheses. The slash symbol is also used when referring to registers or timers that differ on the two machines. For example, Timer A/Timer 1 means that the description applies to Timer A on the Commodore 64 and Timer 1 on the VIC-20. This is also why the microprocessors are referred to as 6510/6502. The meaning of the few exceptions to these rules should be obvious from the context.

While most routines in the Kernal are described in one section that applies to both the 64 and the VIC, there are some routines that are so different that they require individual descriptions.

This book is best used in conjunction with *Mapping the VIC*, by Russ Davies, and *Mapping the Commodore 64*, by Sheldon Leemon, both from COMPUTE! Books. Since we will generally just hop right into the details of a topic, consult the memory map to get a general summary of a routine, and then come back here for the details. The cross reference of routines in sequential address order to page numbers in Appendix A should be useful in this respect.

I want to express my special thanks to Russ Davies for his help and many suggestions. And I appreciate the help given by Stephen Levy, Sheldon Leemon, James Doody, Larry Speth, and Larry Paxman.

This book is dedicated to Grandma Hilda.

## **Chapter 1**

---

# **Interrupts and System Reset**





# Interrupts and System Reset

---

## Starting and Stopping

One of the most important tasks a computer's operating system must be able to handle is startup. When you turn on the computer, you need to be assured that everything is in order. That way, if you make a mistake while programming and cause a *lockup* (where the keyboard will no longer respond), you know you can simply turn the computer off and back on and all will be well again. This is called *resetting* the computer. It is also important to be able to signal the computer to perform some special function, instead of continuing its normal routine. One way to stop the computer's ordinary processing is with an *interrupt*.

When the VIC-20 or Commodore 64 is turned on, circuitry within the computer generates a RESET signal (which is *low*, or zero voltage) that is sent to the RESET pins of the 6510/6502 and the I/O chips (the 6526/6522 and the 6581). When power is first applied, the state of the registers in the I/O chips is random. The RESET signal to the 6526 performs the following initialization sequence: All data port pins are set as inputs, data port registers are cleared to zero, timer control registers are set to disable the timers, timer control latches are changed to all ones, and all other registers are cleared to zero, thus disabling the interrupt output line.

The RESET signal to the 6522 resets the chip as follows: All data port pins are set as inputs, data port registers are cleared to zero, the timers and interrupt output line are disabled. RESET to the 6581 sets all registers to zero and turns off the audio output. Thus, all I/O chips except the VIC chip are placed in a known state with interrupts disabled.

The RESET signal to the 6510/6502 starts the microprocessor's internal reset sequence. After six clock cycles for system initialization, the I flag of the status register is set to disable IRQ interrupts. Then the program counter is loaded from (FFFC), the 6510/6502 RESET vector. On the 64, the program counter is set to FCE2, while the VIC-20 program

counter is set to FD22. These values point to the RESET routines that will be executed on startup.

The system reset software at FCE2/FD22 then disables IRQ interrupts (not really required according to the data sheet since the I flag should already be set to 1); initializes the stack pointer; checks whether an autostart cartridge is present; initializes memory, the RAM Kernal vectors, and the I/O registers (6526/6522, 6567/6560-6561, 6581); sets timer A/timer 1 values and enables interrupts for CIA #1 timer A/VIA #2 timer 1; enables IRQ interrupts; and jumps to a warm start of BASIC. This reset sequence cannot be interrupted since IRQ interrupts have been disabled and since the source of NMI interrupts from CIA #2/VIA #1 has been disabled.

Once the reset sequence ends, the program that receives control (either the built-in BASIC ROM or an autostart cartridge) executes instruction after instruction in its program. If this program execution were all that happened, the results of the program could never be sent to an output device and the program could never receive data from an input device. The I/O interrupt system solves this problem of no communication between the 6510/6502 and the I/O devices such as the keyboard, printer, disk drive, tape drive, and video screen. This chapter discusses general concepts about interrupts. To see how the VIC-20 and 64 NMI and IRQ interrupt handlers actually function, see the detailed descriptions later in the book.

### Interrupt Levels

Large mainframe computers typically have several classes of interrupts such as program check, machine check, I/O, restart, and supervisor call. If you try to execute a program on an IBM System/370 machine and use an undefined opcode, you get a program check interruption for an operation exception. If you try to execute a program on the 6510/6502 with an undefined opcode, the results are not guaranteed. Commodore includes a disclaimer on the data sheets for 6510/6502 stating that they cannot assume liability for the use of undefined opcodes. A few articles and letters have explored these undefined opcodes, such as "Extra Instructions" by Joel Shephard in *COMPUTE!* (October 1983), a letter from Henry Gibbons in *COMPUTE!* (January 1983), and an article by Gary Cordelli, "6502 MPU Hybrid" in *Commander* (December 1982). *Programming the*



*Commodore 64* by Raeto Collin West (from COMPUTE! Books) also has a table of *quasi-opcodes* in the appendix.

Whenever an I/O device needs attention, it generates an interrupt. The 6510/6502 recognizes the interrupt, saves information about the status of the program currently being executed, passes control to the interrupt service routine which services the device which caused the interrupt, and then returns control to the interrupted program. This description is a very general overview of the interrupt-driven Commodore 64 and VIC-20.

The 6510/6502 chips have two interrupt input lines, IRQ and NMI. IRQ and NMI recognize interrupt signals differently. Whenever the IRQ input line is pulled low, the 6510/6502 completes execution of the current instruction. Then the interrupt mask flag (I) of the status register is examined. If I is set to 1, IRQ interrupts are disabled and the IRQ interrupt is ignored. If I is cleared to 0, IRQ interrupts are enabled and the IRQ interrupt sequence begins. The program counter (high byte then low byte) and the status register are pushed onto the stack. Then the interrupt mask flag (I) in the status register is set to 1 to disable further IRQ interrupts so that the IRQ service routine is not itself interrupted by the same IRQ signal. Then the program counter is loaded from the two-byte pointer at (FFFE), the 6510/6502 IRQ vector. The IRQ interrupt handler must clear the source of the interrupt on the I/O chip that produced it before enabling IRQ interrupts again, or the same interrupt would be serviced again. IRQ interrupts for the 6510/6502 are re-enabled (I is cleared to 0) by either the CLI instruction or the RTI instruction. RTI restores the status register to its state at the time the IRQ interrupt was recognized. Since IRQ interrupts had to be enabled for the interrupt to be recognized, the RTI thus re-enables IRQ interrupts.

When the microprocessor's NMI (nonmaskable interrupt) input line has a high-to-low transition, an NMI interrupt occurs. As the name implies, NMI responses cannot be disabled on the 6510/6502. When an NMI interrupt is recognized, the 6510/6502 pushes the program counter and the status register onto the stack. Then the I flag of the status register is set to 1 to disable IRQ interrupts; thus the NMI service routine cannot be interrupted by a signal on the IRQ line. The program counter is then loaded from the two-byte pointer at (FFFA), the 6510/6502 NMI vector. For another NMI interrupt to be

recognized, the NMI input must go high and then back low. Thus, the signal that caused the NMI interrupt is only serviced once. The NMI interrupt handler routine should clear the source of the interrupt on the I/O chip. When the RTI opcode is executed, IRQ interrupts are either enabled or disabled to the same status as when the NMI interrupt occurred.

The time it takes for the 6510/6502 to respond to the NMI or IRQ interrupt signal depends on which instruction is being executed and where in the instruction cycle the interrupt signal occurs. If the last cycle of an instruction is being executed, then the next cycle recognizes the interrupt (no delay). The slowest response to an interrupt occurs with a seven-cycle instruction such as LSR \$2005,X. If the interrupt occurs during the first cycle, this leaves six more instruction cycles to execute before recognizing NMI or IRQ.

### **Servicing Interrupts**

To service an interrupt once it is detected, you must know what caused the interrupt. A method known as polling is used to determine which device caused the interrupt. The NMI and IRQ interrupt handlers interrogate the interrupt flags on the I/O chips to determine which device caused the interrupt. The interrupt handlers check for the interrupt sources in a specific sequence, and thus certain devices or conditions have a higher priority than others and will be serviced first by the interrupt handler. The NMI interrupt handler should make sure all possible sources of the NMI interrupt are serviced before executing RTI. If the NMI interrupt handler does not check all sources (for example, if it services the first source and then does RTI), then if one of the unchecked sources was generating an NMI low signal when RTI is executed the NMI signal is still low. Since NMI must go high and then low for another NMI interrupt to be recognized, you can in effect disable all further NMI interrupts if you don't service all possible sources of NMI.

The IRQ interrupt handler does not have to check all possible sources of IRQ low signals. When the IRQ interrupt handler executes RTI, IRQ interrupts are re-enabled. If one of the active sources of IRQ interrupts was not serviced by the IRQ interrupt handler, this unchecked source will still be holding the IRQ line low upon RTI and thus will cause another IRQ interrupt. The various I/O chips that are polled by the

interrupt handlers mostly have edge-sensitive inputs (this means they detect transitions in voltage) from the I/O devices. When the I/O device or another I/O chip register requests an interrupt, the interrupt condition is latched into an interrupt flag register to allow polling by the NMI or IRQ interrupt handler. If interrupts are enabled for the device or I/O chip register, the output IRQ signal is brought low. This output IRQ signal can be connected to either the NMI or IRQ input pins of the 6510/6502. The NMI or IRQ interrupt handler is then responsible for clearing the interrupt flag during its polling sequence, thus allowing the I/O IRQ output line to return to high. The X register, Y register, and accumulator should be saved by the interrupt service routine upon entry and restored before exiting the handler.

Interrupts from I/O devices or I/O chip registers can occur and set an interrupt flag in the I/O chip interrupt flag register. Rather than have this interrupt passed on to the IRQ or NMI interrupt handler routines by bringing the I/O chip's IRQ output line low, you can write your own interrupt handler that polls the I/O device interrupt flag register. The serial I/O routine uses this technique of having its own polling subroutine for timer interrupts to detect the times for various handshake conditions during serial data transfer. Program 1-1 (for the Commodore 64) also demonstrates this method by polling timer A interrupts from the CIA #1 interrupt data register rather than allowing the timer A interrupt to generate an IRQ interrupt itself.

The 6510/6502 BRK instruction (opcode 00) can be used to generate an IRQ interrupt from a program, rather than from an I/O device. BRK increments the program counter by two and sets the break flag in the status register to 1. The just-modified program counter and status register are then pushed onto the stack, and the program counter is loaded with the IRQ vector at (FFFE). The Kernal interrupt service routine on the 64 and VIC-20 examines the break flag in the copy of the status register that has been pushed onto the stack to detect whether a BRK instruction caused the IRQ interrupt.

### Interrupt Sources

Figures 1-1 and 1-2 show the sources of NMI and IRQ interrupts on the VIC-20 and the order in which the interrupt handlers service the possible sources.

**Figure 1-1. VIC-20 NMI Interrupts**

I/O Interrupt Flag Register	Source	Order of Service	Comments
911D bit 1	VIA#1 CA1 from RESTORE key	1	Test for autostart cartridges; if STOP key is pressed then execute BRK routine
911D bit 6	VIA#1 Timer 1	2	RS-232 send
911D bit 5	VIA#1 Timer 2	3	RS-232 receive
911D bit 4	VIA#1 CB1	4	RS-232 receive

**Figure 1-2. VIC-20 IRQ Interrupts**

I/O Interrupt Flag Register	Source	Comments
912D bit 6	VIA#2 Timer 1	Normal IRQ handler; called every 1/60 sec. to scan keyboard, update jiffy clock, blink cursor, etc.
912D bit 5	VIA#2 Timer 2	Tape IRQ handler, tape write
912D bit 1	VIA#2 CA1	Tape IRQ handler; tape read
912D bit 6	VIA#2 Timer 1	Tape IRQ handler; tape read
912D bit 5	VIA#2 Timer 2	Used by serial I/O

Only one of the timer 1, timer 2, or CB1 interrupts is serviced by any one execution of the VIC-20 NMI interrupt handler. The potential problem mentioned earlier of an unserviced NMI interrupt keeping NMI low and thus disabling further NMI interrupts does not occur, though. Rather, another problem appears. The interrupt enable register is temporarily cleared when servicing either timer 1, timer 2, or CB1 interrupts, resulting in the NMI output line from the VIA going high. So, the next NMI low would cause another interrupt. This interrupt could occur after clearing the timer 1 interrupt flag, but before executing the RS-232 send routine. It could also occur after clearing the timer 2 interrupt flag, but before executing the RS-232 receive routine. Thus nested NMI interrupts for timer 1 and timer 2 could occur and disrupt the RS-232 send/receive timing sequences. Apparently in recognition

## Interrupts and System Reset

of this problem, the NMI interrupt handler was redesigned on the Commodore 64.

Figures 1-3 and 1-4 show the sources of NMI and IRQ interrupts on the Commodore 64 and the order in which the interrupt handlers service the possible sources.

**Figure 1-3. Commodore 64 NMI Interrupts**

I/O Interrupt Flag Register	Source	Order of Service	Comments
DD0D bit 0	CIA#2 timer A*	1	RS-232 send
DD0D bit 1	CIA#2 timer B*	2	RS-232 receive
DD0D bit 4	CIA#2 FLAG1*	3	RS-232 receive
DD0D bit 1	CIA#2 timer B+	4	RS-232 receive
DD0D bit 4	CIA#2 FLAG1+	5	RS-232 receive
Connected directly to 6510 NMI line	RESTORE key	6	Test for autostart cartridge; if STOP key is pressed execute BRK routine

**Figure 1-4. Commodore 64 IRQ Interrupts**

I/O Interrupt Flag Register	Source	Comments
DC0D bit 0	CIA#1 Timer A	Normal IRQ handler; called every 1/60 sec. to scan keyboard, update jiffy clock, blink cursor, etc.
D019 bit 0	6567 raster compare	Determines whether PAL or NTSC video format is being used (called during system reset)
DC0D bit 1	CIA#1 Timer B	Tape IRQ handler; tape write
DC0D bit 4	CIA#1 FLAG1	Tape IRQ handler; tape read
DC0D bit 0	CIA#1 Timer A	Tape IRQ handler; tape read
DC0D bit 1	CIA#1 Timer B	Used by serial I/O

Figure 1-5 shows possible NMI and IRQ interrupt sources that are not serviced by the Kernal's NMI and IRQ interrupt handlers. In order to use these, you must change the vectors to the interrupt handlers, test for the source of the interrupt, and process accordingly.

**Figure 1-5. Interrupts Not Serviced or Enabled by Kernal Routines on Commodore 64 or VIC-20.**

I/O Interrupt Flag Register	Type	Comments
<b>Commodore 64</b>		
D019 bit 1	IRQ	6567 sprite-to-background collision
D019 bit 2	IRQ	6567 sprite-to-sprite collision
D019 bit 3	IRQ	6567 light pen trigger
DC0D bit 3	IRQ	CIA#1 serial port full/empty
DC0D bit 2	IRQ	CIA#1 TOD clock alarm
DC0D bit 4	IRQ	CIA#1 serial bus SRQ in (DC0D bit 4 is serviced for tape read)
DD0D bit 3	NMI	CIA2 serial port full/empty
<b>VIC-20</b>		
911D bit 0	NMI	VIA#1 CA2 tape motor
911D bit 3	NMI	VIA#1 CB2
911D bit 2	NMI	VIA#1 Shift Register
912D bit 0	IRQ	VIA#2 CA2 serial clock out
912D bit 2	IRQ	VIA#2 Shift Register
912D bit 3	IRQ	VIA#2 CB2 serial data out
912D bit 4	IRQ	VIA#2 CB1 serial SRQ in

## An IRQ Application

Program 1-1 illustrates the technique of adding a routine to the beginning of the Commodore 64's normal IRQ interrupt handler. This program fixes the problem that occurs on some older 64s when you are entering characters at the bottom of the screen, press delete, and thus lock up the keyboard. To see if your 64 has the version of the Kernal which includes this bug, go to the bottom of the screen, enter one line of characters and when the cursor wraps to the next line enter a second line of characters. Stop entering characters as soon as the second line scrolls and the cursor returns to the bottom left corner. Then press DEL. The word LOAD is displayed and if a program is in memory it runs. In either case, the keyboard locks up.

Since this lockup apparently is due to destroying the contents of the CIA registers that are located in memory immediately past color RAM, Program 1-1 does not rely on the

contents of the CIA registers. Thus, the timer A interrupt that normally drives the 1/60 second IRQ interrupt is not used. Instead Program 1-1 uses the raster scan interrupt that occurs at twice the normal timer A interrupt rate (120 times per second). This new IRQ wedge (so-called because it is inserted into the normal interrupt execution) then checks the timer A bit in the CIA #1 interrupt data register to see if the keyboard should be scanned and the jiffy clock updated. Thus the routine does not rely on the timer A interrupt, which could be corrupted if the keyboard lockup occurs.

With the safe raster scan interrupt always occurring, the wedge can check to see if any registers in the CIA that should always contain one fixed value have been modified. If they have, then it is likely that the keyboard lockup has occurred, and in this case the routine that reinitializes the CIA registers is called. The program demonstrates that your Commodore 64 can be driven by IRQ interrupts from the VIC chip rather than from IRQ interrupts from timer A on the CIA chip. Jim Butterfield's article "Son of Split Screens" in *COMPUTE!'s First Book of Commodore 64* is an excellent model on which this program is based.

### Program 1-1. Keyboard Unlock Routine

```
1000 .OPT NOERRORS,LIST,GENERATE
1010 ;
1020 ; PROGRAM TO SETUP THE NEW IRQ
1030 ; INTERRUPT HANDLER
1040 ; TO EXECUTE THIS PROGRAM EITHER
1050 ; SYS 52752 FROM BASIC OR
1060 ; G CE10 FROM ML MONITOR
1070 ; AFTER THE PROGRAM IS LOADED
1080 ;
1090 ;
1100 ;
1110 ;
1120 * = $CE10
1130 SEI
1140 LDA #$00
1150 STA $0314 ; RESET IRQ
1160 LDA #$CF ; VECTOR (0314)
1170 STA $0315 ; TO $CF00
1180 LDA #$7F ; TURN OFF TIMER A INTERRUPTS
1190 STA $DC0D
1200 LDA $D011 ; TURN OFF HIGH
1210 AND #$7F ; BIT OF RASTER
1220 STA $D011 ; SCAN REGISTER
1230 LDA #$01 ; SET RASTER
```

## Interrupts and System Reset

```

1240      STA $D012 ; REGISTER TO 1
1250      LDA $D01A ; INTERRUPT ENABLE REGISTER
1260 ;                               FOR 6567
1270      ORA #$81 ; ENABLE 6567 RASTER INTERRUPT
1280      STA $D01A ; WHEN RASTER COUNT = RASTER REG
1290      CLI ; RETURN TO CF00 NEXT IRQ
1300      RTS
1310 ;
1320 ;
1330 ;
1340 ;
1350 ;
1360 ;
1370 ;
1380 ;
1390 ;
1400 ;
1410 ; FOLLOWING IS THE NEW IRQ INTERRUPT
1420 ; HANDLER AT CF00 THAT USES THE RASTER INTERRUPTS
1430 ; RATHER THAN TIMER A INTERRUPTS
1440 ; TO DRIVE IRQ INTERRUPT PROCESSING
1450 ;
1460 ; THIS INTERRUPT HANDLER RECOVERS
1470 ; FROM A KEYBOARD LOCKUP
1480 ; BY CHECKING FOR THE WIPEOUT OF
1490 ; THE CIA REGISTERS THAT OCCURS
1500 ; DURING KEYBOARD LOCKUP AND
1510 ; THEN RESTORES THE CIA REGISTERS
1520 ; BACK TO THEIR CORRECT SETTINGS
1530 ;
1540 ;
1550      * = $CF00
1560      LDA #$01
1570      STA $D019 ; CLEAR RASTER LATCH
1580      LDA $DC00 ; CHECK CIA 1 DATA PORT A
1590      CMP #$7F ; FOR CORRECT COLUMN SETUP
1600      BNE LOCKUP
1610      LDA $DC02 ; ALSO CHECK DATA DIRECTION OF
1620 ;                               PORT A
1630      CMP #$FF ; SHOULD BE ALL OUTPUTS
1640      BNE LOCKUP
1650 ; IF THE KEYBOARD IS NOT LOCKED UP
1660 ; THEN SEE A TIMER A INTERRUPT HAS
1670 ; OCCURRED AND SET LATCH IN INTERRUPT DATA REGISTER
1680      LDA $DC0D ; CIA 1 INTERRUPT DATA REG
1690      AND #$01 ; LEAVE ONLY TIMER A BIT
1700      BEQ NOTA ; BEQ IF NO TIMER A INTERRUPT
1710      JMP $EA31 ; IF TIMER A THEN DO NORMAL IRQ
1720 ;
1730 ;
1740 ; RECOVER FROM KEYBOARD LOCKUP
1750 LOCKUP JSR $FF84 ; IOINIT - BUT RENABLES
1760 ;                               TIMER A IRQ
1770      LDA #$7F ; SO TURN OFF TIMER A AGAIN

```



## Interrupts and System Reset

```
1780          STA $DC0D
1790 ;
1800 ; IF NO TIMER A INTERRUPT OR IF
1810 ; KEYBOARD LOCKED UP
1820 NOTA     PLA          ; RESTORE REGISTERS
1830          TAY          ; SAVED AT ENTRY
1840          PLA          ; TO IRQ
1850          TAX
1860          PLA
1870          RTI
1880          .END
```



## **Chapter 2**

---

# **System Reset**



# System Reset

---

This chapter is a detailed look at the reset routine in the 64 and VIC, which is activated when the power is turned on.

## System Reset

FCE2/FD22-FD01/FD3E

### Called by:

The program counter loads the two-byte RESET vector at (FFFC) when RESET input to the 6510/6502 goes low. This routine controls the software reset, and calls subroutines to perform some aspects of initialization. From this routine, control is passed to either BASIC's cold start routine or to the cold start routine of an autostart cartridge, if one is present.

### Entry conditions:

IRQ interrupts are disabled by hardware. I/O chips should have accepted the system RESET signal and performed their own internal reset functions.

### Operation:

1. Disable IRQ interrupts.
2. Initialize stack pointer to \$FF. Page one of RAM (0100-01FF) is the stack, thus the stack pointer points to 01FF.
3. Clear the decimal flag in the status register so that addition and subtraction operations produce proper binary results.
4. JSR FD02/FD3F to check for autostart cartridge. Return with Z (zero) bit of status register set to 1 if an autostart cartridge is present and has proper autostart character sequence.
5. If status register Z flag was set to 0 following the check for an autostart cartridge (step 4), then BNE to step 7.
6. If Z=1 then an autostart cartridge was found. In this case, immediately pass control to the autostart cartridge cold start routine by JMP(8000)/JMP(A000).

## System Reset

*The following steps 7–13 apply only to the 64:*

7. If the autostart sequence was not found, then store the X register (which now has a value from 1–5) into D016, the VIC chip control register, thus forcing bit 5, the reset bit, to be set to 0.
8. JSR FDA3 to initialize the CIA registers, the 6510 built-in I/O port and data direction registers, and SID registers; start CIA #1 timer A, and set the serial clock output line high.
9. JSR FD50 to initialize memory pointers.
10. JSR FD15 to initialize the Kernal RAM vectors that start at (0314).
11. JSR FF5B to initialize the VIC-II chip registers, set the PAL/NTSC flag and set timer A to a value based on this flag.
12. Enable IRQ interrupts.
13. JMP(A000) to BASIC's Cold Start routine.

*The following steps 7–12 apply only to the VIC-20:*

7. JSR FD8D to initialize memory pointers.
8. JSR FD52 to initialize the Kernal RAM vectors that start at (0314).
9. JSR FDF9 to initialize the 6522 registers.
10. JSR E518 to initialize the VIC chip registers. (See chapter 7).
11. Enable IRQ interrupts.
12. JMP(C000) to BASIC's cold start routine.

### **Test for Autostart Cartridge FD02/FD3F–FD0F/FD4C**

#### **Called by:**

JSR at FCE7/FD27 in System Reset, JSR at FE56/FD3F in NMI Interrupt Handler.

This routine checks for the characters CBM80 at location 8004 on the 64 or the characters A0CBM at location A004 on the VIC-20. In both of these sequences, the "CBM" characters have the most significant bit set to 1. If this sequence is not found, then the status register Z flag is set to 0 for a subsequent BNE. If the sequence is found, then Z is set to 1 and the system reset routine or the NMI interrupt handler will JMP(8000)/JMP(A000).

### Operation:

1. Initialize X register to 5.
2. Load accumulator with the value at  $FD0F+X/FD4C+X$ .
3. Compare this value loaded into the accumulator with the value at address  $8003+X/A003+X$ . The compare sets the Z flag to 0 if the values are not equal.
4. If not equal, then RTS with the Z bit set to 0 (BNE condition).
5. If equal, then decrement X register.
6. If X register is now 0, then the autostart sequence characters exist; return with the Z bit of status register set to 1.
7. If X register is not yet 0, then branch to step 2.

### Initialize Memory Pointers (64)

#### FD50-FD9A

#### Called by:

JSR at FCF5 in System Reset, JMP from Kernal RAMTAS vector at FF87.

This routine initializes memory locations in pages 0, 2, and 3 of memory and sets pointers to the start of RAM, the end of RAM + 1, and the start of the tape buffer.

### Operation:

1. For  $Y = \$0-FF$ , store 0 at address  $0002+Y$ ,  $0200+Y$ , and  $0300+Y$ , thus storing zero into locations 0002-0101, 0200-03FF. Note that STA  $\$0002,Y$  (absolute addressing indexed with Y) is used to clear zero page. If STA  $\$02,X$  (zero page addressing indexed with X) had been used, then locations 0 and 1 would also have been cleared.
2. Set pointer to tape buffer (B2) to the value  $\$033C$ .
3. Set RAM test pointer (C1) to start at 0400. Set Y register to  $\$00$ . The Y register is used during RAM test to increment through each page of memory.
4. Save the current contents of the RAM location in the X register. Store  $\$55$  into this RAM location and then compare the contents of the location to  $\$55$ . If not equal, then this is not RAM, so branch to step 6. If this test succeeds, then ROL, store the accumulator contents (now  $\$AA$ ) at the same location and compare to  $\$AA$  to see if this is a RAM location. If this test of RAM fails, branch to step 6.

5. After each successful test of a RAM location, restore the original contents that have been saved in the X register and increment the Y register. When the Y register rolls over to zero, then increment C2, the page being tested. Loop to step 4.
6. When the first non-RAM location is found, note its location. TYA and TAX to set the X register with the offset within the page of memory, and LDY C2 retrieves the page of memory.
7. CLC and JSR FE2D to set the pointer to the end of RAM + 1 in (0283).
8. Set the pointer to the start of RAM, (0281), to \$0800.
9. Set the screen memory page number pointer at 0288 to \$04.

### **Initialize Memory Pointers (VIC-20) FD8D-FDFD**

#### **Called by:**

JSR at FD2F in System Reset.

This routine initializes memory in pages 0, 2, and 3. It also tests for the start and end of RAM. RAM must start between 0400 and 1000 inclusive, and RAM must not end before 2000. If these requirements are not met, then the initial BASIC-bytes-free message does not appear. Instead, the screen remains blank white with a blue border. No particular error message is written to the screen.

Screen memory is located in different areas depending on how much RAM the system contains. If the end of RAM + 1  $\geq$  2100, then the screen memory page is initialized to \$10; otherwise, the screen memory page is initialized to \$1E.

The pointer to the tape buffer (B2) is initialized to 033C.

Exit from this routine is made by a JMP to the routine to set the top of memory pointer which then does an RTS. If RAM is not located at the expected areas, then this routine does not exit: an infinite loop of initializing the VIC chip registers is executed.

#### **Operation:**

1. For X = 0 to \$FF, store 0 at address 00+X, 0200+X, and 0300+X; thus pages 0, 2, and 3 are initialized to all zeros.
2. Set pointer to tape buffer (B2) to 033C.
3. Set RAM test pointer (C1) to start its test at 0400.
4. Increment the RAM test pointer.



## System Reset

5. JSR FE91 to test the location pointed to by (C1) to see if it is a RAM location. Upon finding the first RAM location, see if the start of RAM is  $< 1100$ . If not, display the blank error screen in an infinite loop. The infinite loop is: FDEB JSR E5C3; FDEE JMP FDEB.
6. If the location was non-RAM, fall through to step 7. If RAM, loop to step 4.
7. Upon falling through (non-RAM), see if this first non-RAM location is  $< 2000$ .
8. If so, then display the error screen in the infinite loop mentioned in step 5.
9. Was the end of RAM  $+ 1 \geq 2100$ ?
10. If no, then set screen memory page pointer (0288) to 1E00, then JMP FE7B to set the top of memory  $+ 1$  (0283) to 1E00 and RTS.
11. If the end of RAM  $+ 1$  was  $\geq 2100$ , then set the screen memory page pointer (0288) to 1000, reset the pointer to the start of memory (0282) to 1200, and set the pointer to the end of memory  $+ 1$  (0283) to the first non-RAM location found.

### Test for RAM Byte (VIC-20) FE91-FEA8

#### Called by:

JSR at FDB5 in Initialize Memory Pointers.

A byte of memory is tested to see if it is RAM by attempting to store a value in the byte. If the value can be stored, then this byte is RAM. The status register carry flag is clear on exit if this byte is non-RAM, while the carry flag is set if the byte is RAM. The RAM test is nondestructive since the original value of the byte to be tested is saved on entry to the routine and restored on exit. By trying to store two different values into the byte being tested, you not only perform a better validation that the memory element is functioning properly, but you also protect yourself from setting the pointers incorrectly if by rare chance the values in a ROM location matched exactly with the value you were trying to store.

#### Operation:

1. Load accumulator with the value in the byte of memory to be tested.

## System Reset

---

2. TAX to temporarily store the original value.
3. Store \$55 in the byte.
4. Compare the value of the byte to \$55 to see if \$55 was actually stored.
5. If not, then clear the carry and branch to step 9.
6. If it was stored, then rotate right (ROR) into the carry (\$55 rotated right sets the carry).
7. Store value in accumulator into the byte of memory being tested once again.
8. Is the value stored equal to the accumulator? If not, then clear the carry.
9. Restore original value of the byte from the X register.
10. RTS with carry clear if non-RAM or carry set if RAM.

### **Initialize Kernal RAM Vectors** **FD15/FD52-FD2F/FD6C**

#### **Called by:**

JSR at FCF8/FD32 in System Reset, JSR at FE66/FED2 in BRK Interrupt Handler, JMP from Kernal RESTOR vector at FF8A; alternate entry at FD1A/FD57 by JMP from Kernal Vector at FF8D.

This routine initializes the Kernal vectors at (0314)–(0332). The vector table in ROM at FD30/FD6D is used for the Kernal vectors when the routine is entered from the System Reset, BRK, or RESTOR routines.

If you enter this routine at the alternate entry point and with the status register carry bit clear, then the vectors from (0314)–(0322) will instead be loaded from memory beginning at the address specified by the contents of the X and Y registers. This allows you to create your own vector table. If you want to use your own I/O routines instead of the system defaults for any of the routines with RAM vectors at 0314–0332, you must change the vector either directly or through using the Kernal Vector routine. You must reset your own vectors after each RESET or NMI interrupt caused by the RESTORE key, as the default vectors are reloaded in these situations.

This routine can also be used to copy the contents of the Kernal RAM vectors. If you enter the routine at the alternate point with the carry bit set, the contents of (0314)–(0332) are stored at the location specified by the contents of the X and Y registers.

## System Reset

### Entry requirements (for FD1A/FD57):

The carry bit should be set or clear, depending on the function desired:

Set the carry bit to store the RAM vectors at (0314)–(0332) at the location pointed to by the X and Y registers. X should hold the low byte of the storage address and Y should hold the high byte.

Clear the carry bit to load the RAM vectors at (0314)–(0332) from the location pointed to by the X and Y registers. X should hold the low byte of the storage address and Y should hold the high byte.

### Exit conditions:

The default settings for the vectors are established as shown below:

#### Default Kernal RAM Vectors

Vector	64	VIC	Function
(0314)	EA31	EABF	IRQ interrupt handler
(0316)	FE66	FED2	BRK interrupt handler
(0318)	FE47	FEAD	NMI interrupt handler
(031A)	F34A	F40A	OPEN
(031C)	F291	F34A	CLOSE
(031E)	F20E	F2C7	Set input device
(0320)	F250	F309	Set output device
(0322)	F333	F3F3	Reset default I/O
(0324)	F157	F20E	Input from device
(0326)	F1CA	F27A	Output to device
(0328)	F6ED	F770	Test STOP key
(032A)	F13E	F1F5	Get from keyboard
(032C)	F32F	F3EF	Close files
(032E)	FE66	FED2	Unused (points to BRK handler)
(0330)	F4A5	F549	LOAD
(0332)	F5ED	F685	SAVE

### Operation:

1. FD15/FD52: Load X and Y with the address of the default vector table, FD30/FD6D, and clear the carry.
2. Store X and Y at (C3), the base address of the vector table.
3. If carry is clear then use the table addressed by (C3) and load the RAM vectors at (0314)–(0332) from this table.
4. If carry is set, then read the RAM vectors at (0314)–(0332) and store them at the location address by (C3).

### Initialize I/O Chips (64) FDA3-FDF8 and FF6E-FF80

#### Called by:

JSR at FCF2 in System Reset, JSR at FE69 in BRK Interrupt Handler, JMP from Kernal IOINIT vector at FF84; alternate entry at FDDD by JMP at FF6B in Initialize VIC Chip and Set PAL/NTSC Flag, JSR at FCA5 in Reset I/O Registers and Restore IRQ Vector.

This routine initializes the two CIA chips, the SID chip, and the 6510 I/O port; starts CIA #1 timer A (which is used to trigger the system IRQ every 1/60 second); and sets the serial bus clock output line. The alternate entry point is used to set the correct value in timer A based on the PAL/NTSC flag.

#### Operation:

1. Store \$7F in DC0D, CIA #1 interrupt control register, thus clearing all interrupt masks to disable interrupts from CIA #1.
2. Store \$7F in DD0D, CIA #2 interrupt control register, thus clearing all interrupt masks to disable interrupts from CIA #2.
3. Store \$7F in DC00, CIA #1 data port A to set the keyboard column values for the keyboard scan.
4. Store \$08 in DC0E, CIA #1 control register A, and in DD0E, CIA #2 control register A, to set the values shown below for both CIA chips:

#### Initial CIA Control Register A Settings

- Bit 7 = 0: Time-of-day clock frequency, 60 Hz
- Bit 6 = 0: Serial port I/O mode is input
- Bit 5 = 0: Timer A counts system  $\phi$ 2 clock pulses
- Bit 4 = 0: Don't force a load of timer A
- Bit 3 = 1: Timer A run mode is one-shot
- Bit 2 = 0: Timer A output mode to PB6 is pulse
- Bit 1 = 0: No timer A output on PB6
- Bit 0 = 0: Stop timer A

5. Store \$08 to DC0F, CIA #1 control register B, and to DD0F, CIA #2 control register B, to set the values shown below for both CIA chips:

#### Initial CIA Control Register B Settings

- Bit 7 = 0: Set time-of-day clock
- Bits 6-5 = 00: Timer B counts system  $\phi$ 2 clock pulses
- Bit 4 = 0: Don't force a load of timer B
- Bit 3 = 1: Timer B run mode is one-shot
- Bit 2 = 0: Timer B output to PB7 is pulse

## System Reset

Bit 1 = 0: No timer B output on PB7

Bit 0 = 0: Stop timer B

6. Store \$00 in DC03, CIA #1 data direction register, for port B and in DD03, CIA #2 data direction register for port B. This defines all bits in DC01, CIA #1 data port B, and DD01, CIA #2 data port B, as inputs.
7. Store \$00 in D418, a 6581 SID chip register, to turn the output volume off so that the chip will be silent following initialization.
8. Store \$FF in DC02 CIA #1 data direction register for port A, thus setting all bits in DC00, CIA #1 data port A, as outputs.
9. Store \$07 in DD00, CIA #2 data port A, and \$3F in DD02, CIA #2 data direction register for port A, to initialize port A for the functions shown below:

### Initial CIA #2 Data Port A Definitions

Bit 7 = 0: Serial bus data input

Bit 6 = 0: Serial bus clock input

Bit 5 = 0: Serial bus data output

Bit 4 = 0: Serial bus clock output

Bit 3 = 0: Serial bus attention signal output

Bit 2 = 1: RS-232 data output

Bits 1-0 = 11: VIC chip memory bank select

10. Store \$E7 in 01, the 6510's built-in I/O port, and \$2F in 00, the data direction register for that port, to initialize the port for the functions shown below:

### Initial 6510 I/O Port Values

bit 7 = 1: Set for input, but no corresponding 6510 pin exists

bit 6 = 1: (In) Set for input, but no corresponding 6510 pin exists

bit 5 = 1: Tape motor control output (1 = motor on)

bit 4 = 0: Tape button sense input line

bit 3 = 0: Tape write output line

bit 2 = 1: CHAREN output—select character ROM or I/O devices at D000-DFFF (1 = I/O devices)

bit 1 = 1: HIRAM output—controls whether the 64 sees RAM or Kernal ROM in address space E000-FFFF (1 = Kernal; if RAM is selected here, then BASIC ROM is also replaced by RAM)

bit 0 = 1: LORAM output—controls whether the 64 sees RAM or BASIC ROM in address space A000-BFFF (1 = BASIC)

## System Reset

11. FDDD: See if 02A6, the PAL/NTSC flag, is 0 (NTSC) or 1 (PAL). If 02A6 indicates PAL, then initialize timer A on CIA #1 by writing \$25 to DC04, timer A low byte latch, and \$40 to DC05, timer A high byte latch, thus setting a latch value of \$4025 for timer A.  
If 02A6 indicates NTSC, then initialize timer A on CIA #1 by writing \$95 to DC04, timer A low byte latch, and \$42 to DC05, timer A high byte latch, thus setting a latch value of \$4295 for timer A.
12. JMP to FF6E for the remainder of this routine.
13. FF6E: Store \$81 to the write-only DC0D CIA #1 interrupt control register, enabling interrupts and setting the mask for CIA #1 timer A interrupts (bit 0). Thus, timer A interrupts will now generate an interrupt request in the read-only interrupt control data register at DC0D and will also set the timer A latch in the read-only data register.
14. Load accumulator from DC0E, CIA #1 control register A. AND with \$80, setting all bits to zero except for bit 7. Bit 7 is left unchanged to maintain the setting for the frequency of the time-of-day clock, 50 Hz(1) or 60 Hz(0). Then ORA \$11 to force the timer A latch values to be loaded into the timer A counter and to start timer A countdown. Store the accumulator contents back into DC0E to initiate the process.
15. JMP EE8E to load accumulator from DD00, CIA #2 data port A, then ORA \$10, and store the accumulator back into DD00. This sets the serial bus clock output line to 1. The output line then goes through an inverter so the actual voltage on the CLK line at the serial port is low.

### Initialize VIA Registers (VIC-20) FDF9-FE48

#### Called by:

JSR at FD35 in System Reset; JSR at FED5 in BRK Interrupt Handler; alternate entry at FE39 from JSR at FCE3 in Reset I/O Registers and Restore IRQ Vector.

This routine initializes the registers of the 6522 VIA chips in the VIC, and sets the CB1 and CB2 control lines. It also sets data direction for ports A and B, and disables all interrupts from the VIAs except for timer 1 interrupts on VIA #2 (used to trigger the system IRQ interrupt every 1/60 second). Timer 1 on VIA #2 is initialized to 17,033 (decimal).

### Operation:

1. Disable all interrupts from VIA #1 and VIA #2 by storing \$7E in 911E and 912E, the interrupt control registers.
2. Store \$40 in the auxiliary control registers for VIA #1, 911B, and VIA #2, 912B, setting the values shown below:

#### Initial Auxiliary Control for VIA Chips

Bits 7–6 = 01: Timer 1 in continuous free-running mode, with output on PB7 disabled

Bit 5 = 0: Timer 2 is interval timer in one-shot mode

Bits 4–2 = 000: Shift register disabled

Bit 1 = 0: Data port B reflects pin values, not latched values

Bit 0 = 0: Data port A reflects pin values, not latched values

3. Store \$FE in 911C, VIA #1 peripheral control register, giving the values in the table below:

#### Initial VIA #1 Peripheral Control

Bits 7–5 = 111: CB2 is output, held high

Bit 4 = 1: CB1 interrupt flag is set on a low-to-high transition of CB1 input

Bits 3–1 = 111: CA2 is output, held high

Bit 0 = 0: CA1 interrupt flag is set on a high-to-low CA1 input transition

4. Store \$DE in 912C, VIA #2 peripheral control register, giving the values listed here:

#### Initial VIA #2 Peripheral Control

Bits 7–5 = 110: CB2 is output, held low

Bit 4 = 1: CB1 interrupt flag is set on a low-to-high transition of CB1 input

Bits 3–1 = 111: CA2 is output, held high

Bit 0 = 0: CA1 interrupt flag is set on a high-to-low CA1 input transition

5. Store \$00 in 9112, VIA #1 port B data direction register, to set all bits in port B as inputs.
6. Store \$FF in 9122, VIA #2 port B data direction register, to set all bits in port B as outputs.
7. Store \$00 in 9123, VIA #2 port A data direction register, to set all bits in port A as inputs.
8. Store \$80 in 9113, VIA #1 port A data direction register, to set bits 0–6 in port A as inputs and bit 7 as an output.
9. Store \$00 into 911F to clear the nonhandshaking data port A for VIA #1.
10. JSR EF84 to set VIA #2 peripheral control register to hold CA2 low on VIA #2.

11. Store \$82 in 911E, VIA #1 interrupt enable register, to enable interrupts from CA1 for VIA #1 (the RESTORE key interrupt).
12. JSR EF8D to set VIA #2 peripheral control register to hold the CA2 high on VIA #2 (the serial bus clock output line). Since interrupts are not enabled for a CA2 active transition on VIA #2, this change on CA2 of VIA #2 does not cause an interrupt.
13. FE39: Enable timer 1 interrupts on VIA #1 by storing \$C0 in 912E, VIA #2 interrupt enable register. Thus, the 1/60-second IRQ interrupts from timer 1 are now enabled.
14. Initialize timer 1 on VIA #2. Store \$89 in 9124, timer 1 low latch during a write. Store \$42 in 9125, timer 1 high latch. This store into the timer 1 high latch also triggers the following operations: timer 1 high latch is transferred to timer 1 high counter; timer 1 low latch is transferred to timer 1 low counter. Timer 1 interrupt flag is reset. Since timer 1 is in continuous or free-running mode, it is already counting, and this operation resets the count to \$4289. Since timer 1 interrupts are enabled, when timer 1 counts down to zero, a timer 1 interrupt is generated. Instead of continuing to count down from zero after an interrupt, the contents of the timer 1 latches (which have been set to \$4289 by this step) are transferred to the timer 1 low count and high count bytes, and the countdown continues from this latched value. Thus, timer 1 is set to always produce an IRQ interrupt at the same time interval. The concept of latches and counters is difficult to grasp, but essential if you try to use the timers in your own programs.

### **Initialize VIC-II Chip and Set PAL/NTSC Flag (64) FF5B-FF6D**

#### **Called by:**

JSR at FCFB in System Reset.

This is an interesting routine that allows the 64 to detect whether its VIC-II chip's output is set up for PAL or NTSC video format, allowing software adjustment of any values that must be modified on the two systems. This routine is only found in the ROMs of newer 64s.



## System Reset

---

1. JSR E518 (see Screen section) to set VIC-II chip registers, blank the screen, set the cursor pointers, and initialize the screen line link table.
2. Load accumulator from D012, thus reading the raster register (eight low order bits of raster line).
3. If accumulator is nonzero, branch to step 2. Loop until the raster count goes through zero one time.
4. Load accumulator from D019, the VIC chip interrupt flag register.
5. AND \$01 to use setting of raster compare IRQ flag.
6. Store accumulator into 02A6, the PAL/NTSC flag. This stores a 0 if the 64 is using NTSC video, or stores a 1 for PAL video. Bit 7 of the raster register D011 and bits 0-7 of D012 were initialized to \$0137 (311 decimal). Since NTSC video generates only 262 raster lines while PAL video generates 312 raster lines, this raster compare value is reached (and the IRQ raster interrupt flag set) only if using PAL video.
7. JMP FDDD to set CIA #1 timer A value based on the PAL/NTSC flag in 02A6.



## Chapter 3

---

# NMI Interrupts



# NMI Interrupts

---

The two-byte NMI interrupt vector is located at (FFFA). This is a design feature of the 6510/6502 microprocessor. Whenever an NMI interrupt occurs, the address contained in FFFA and FFFB in low byte, high byte format is loaded into the program counter. In VIC and 64 ROM, the value is FE43/FEA9, and, when this routine is executed, IRQ interrupts are disabled, followed by JMP (0318). The default values loaded into locations 318 and 319 during system initialization are FE47/FEAD, which redirect the routine immediately back to the NMI handler in ROM. The indirect jump through the RAM vector allows you to wedge into the NMI handling process by changing the values in 318 and 319 if you wish to add your own NMI routines.

The 64 routine determines whether the interrupt was caused by the RESTORE key, FLAG line, timer A, or timer B.

For the VIC, it determines whether the interrupt was caused by the RESTORE key, timer 1, timer 2, or CB1.

If any of these interrupts has occurred, specific interrupt handling for that condition is performed.

The discussion of RS-232-C routines also treats NMI interrupts, since one main function of NMI interrupts on the 64 and VIC is to handle RS-232-C input/output.

The 64 NMI interrupt handler is considerably rewritten from the VIC version. The VIC allows nested NMI interrupts to occur, which you generally don't want in your NMI interrupt handler.

One somewhat unimportant fact about NMI interrupts is that holding down the RESTORE key is all that is needed to pass control to an autostart cartridge, while both RESTORE and RUN/STOP must be held down to pass control to BASIC.

## **NMI Interrupt Handler Jump FE43/FEA9-FE46/FEAC**

### **Called by:**

Program counter loads NMI vector when the 6510/6502 NMI line goes from high to low.

Whenever an NMI interrupt occurs, the NMI vector in (FFFA), which points to FE43/FEA9, is loaded into the pro-

## NMI Interrupts

gram counter, which causes the routine to be executed. First, IRQ interrupts are disabled so that NMI handling will not be interrupted by IRQ interrupts. Then JMP (0318) to the NMI interrupt handler, whose default is FE47/FEAD.

### Operation:

1. Disable IRQ interrupts.
2. JMP (0318), with a default of FE47/FEAD.

## NMI Interrupt Handler (64)

### FE47-FEC1

#### Called by:

Indirect JMP through (318) at FE44 in NMI Interrupt Handler Jump.

On the 64, NMI interrupts can be caused by a high-to-low transition on the NMI line from pin D of the expansion port, from the IRQ output of CIA #2 as the result of five possible interrupt conditions on that chip (FLAG, shift register, time-of-day clock alarm, timer B, or timer A), or from the RESTORE key. The handler routine tests only FLAG, timer B, timer A, and the RESTORE key (checking for the presence of an autostart cartridge). To service the shift register or TOD alarm from CIA #2, you must check these conditions yourself, possibly by modifying the RAM vector to execute your NMI routine which checks for the TOD alarm interrupt or the shift register interrupt before returning to the normal NMI interrupt handler in ROM.

The timer A, timer B, and FLAG interrupts are used in RS-232-C communications. Timer A interrupts control RS-232-C transmission. Timer B interrupts control reception of each bit for RS-232-C. FLAG interrupts detect when reception is to begin and initiate the input for each byte, initializing timer B and enabling the timer B interrupt while disabling further FLAG interrupts while the byte is being received.

### Operation:

1. Save the accumulator, X register, and Y register on the stack.
2. Store \$7F into DD0D, CIA #2 interrupt control register, thus clearing all mask bits in the register and disabling all further interrupts from CIA #2.

## NMI Interrupts

3. Load Y register from DD0D to determine the source of the NMI interrupt.
4. BMI to step 10 if there were interrupts from CIA #2. Bit 7 in DD0D is 1 if an interrupt occurred and the mask for that interrupt was enabled.
5. If CIA #2 was not the source of the interrupt, then the source is the RESTORE key or the NMI line from the expansion port. The RESTORE key on the 64 passes through a timer chip directly to the NMI line to the 6510, unlike the RESTORE key on the VIC which goes to a VIA chip.  
JSR FD02 to test for an autostart cartridge, and if one is present (or if the identifier bytes are found) then JMP (8002) to the cartridge's warm start routine.
6. JSR F6BC to scan the keyboard. The 64, unlike the VIC, does not increment the jiffy clock during an NMI interrupt.
7. JSR FFE1 to test for the STOP key.
8. If the STOP key was detected, then fall through to step 9; otherwise, branch to step 10.
9. FE66: This is also the BRK routine.  
JSR FD15 to initialize the Kernal RAM vectors.  
JSR FDA3 to initialize the I/O chips.  
JSR E518 to initialize the VIC chip register, blank the screen, and create the screen line link table.  
JMP(A000) to do a warm start of BASIC.
10. FE72: Step 10 is reached if one of the following is true: the interrupt is from a CIA #2 source; the interrupt is due to RESTORE, but not STOP-RESTORE, being pressed; or the interrupt is from the expansion port and no autostart cartridge is present.  
Transfer the contents of the Y register, which contains the value from DD0D, CIA #2 interrupt control register, reflecting the source of an NMI interrupt on CIA #2, to the accumulator.
11. AND 02A1, the RS-232-C activity flag.
12. TAX.
13. AND \$01. 02A1 has bit 0 on if transmission has started, and DD0D bit 0 = 1 if a timer A interrupt occurs.
14. If not transmitting or if no timer A interrupt, branch to step 23.

## NMI Interrupts

15. If RS-232-C transmission is active and a timer A interrupt has occurred, then:
  - LDA DD00, CIA #2 data port A.
  - AND \$FB to clear bit 2, the RS-232-C data output, to 0.
  - ORA B5, which contains the next bit to be transmitted.
  - STA DD00, thus transmitting the next bit through CIA #2 data port A.
16. TXA to restore the AND of DD0D and 02A1.
17. AND \$12 to test for FLAG or timer B interrupt.
  - BEQ to step 21 if neither a FLAG or timer B interrupt is pending. Thus, the 64 correctly checks the various possible sources of NMI interrupts, rather than just servicing the first one and then exiting.
18. If either a timer B or a FLAG interrupt occurred, then
  - AND \$02 to test for a timer B interrupt. BEQ to step 20 if not a timer B interrupt, which means the branch is taken if a FLAG interrupt is pending.
19. If timer A and B interrupts occurred at the same time, then the interrupt handler correctly services both interrupts.
  - JSR FED6 to read the next RS-232-C input bit and to add the bit to the current byte being received, or to see if the bit is a start, stop, or parity bit.
  - JMP FE9D to step 21.
20. For a FLAG interrupt:
  - JSR FF07 to set timer B and to reverse 02A1 bits 1 and 4.
21. For either timer B or FLAG interrupt, JSR EEBB to prepare the next bit to transmit.
22. JMP FEB6 to step 27.
23. Branch here from step 14 if this NMI interrupt was from CIA #2, but either RS-232-C transmission was not active or a timer A interrupt did not occur.
  - See if the NMI interrupt was due to timer B interrupt. If not, branch to step 25.
24. For a timer B interrupt, JSR FED6 to read the next RS-232-C input bit, and to add the bit to the current byte being received, or to see if the bit is a start, stop, or parity bit.
  - JMP FEB6 to step 27.



## NMI Interrupts

25. See if the NMI interrupt was due to a FLAG interrupt. If not, branch to step 27.
26. For a FLAG interrupt, JSR FF07 to set timer B.
27. FEB6: LDA 02A1 then STA DD0D thus resetting the CIA interrupt control register to enable the same interrupts that were enabled at entry to the NMI interrupt handler (unless a FLAG interrupt occurred, in which case timer B interrupts are now enabled and FLAG interrupts disabled).
28. Restore Y register, X register, and accumulator from the stack.
29. RTI to return from the interrupt handler.

### **NMI Interrupt Handler—Timer B Service (64) FED6–FF06**

#### **Called by:**

JSR at FE94 and FEA8 in NMI Interrupt Handler.

#### **Operation:**

1. LDA DD01, which contains the RS-232-C data bit received in bit 0.  
    AND \$01.  
    STA A7 to save the received bit.
2. Reset timer B latches from the current value of timer B, minus \$1C, plus the bit time in (0299). Store \$11 in DD0F to force a load of timer B and to start timer B.
3. Restore DD0D, CIA #2 interrupt control register, from 02A1, thus restoring all interrupts that were enabled upon entry to the NMI interrupt handler.
4. Reset timer B latches DD06 and DD07 to \$FFFF, which does not affect the timer B countdown already in progress.
5. JMP EF59 to either store the bit received (held in bit 0 of A7) as a data bit in the current byte being received, or to test for a start, stop, or parity bit.

### **NMI Interrupt Handler—Start Timer B for FLAG NMI (64) FF07–FF2D**

#### **Called by:**

JSR at FE9A and FEB3 in NMI Interrupt Handler.

## NMI Interrupts

### Operation:

1. Store the contents of 0295 in DD06 and the contents of 0296 in DD07, thus setting the latches for timer B. Locations 0295 and 0296 were set from the baud rate tables when the RS-232-C channel was opened.
2. Store \$11 in DD0F to force a load of timer B from the latched values and to start timer B.
3. EOR 02A1 with \$12 and store the result in 02A1, thus reversing the values in the FLAG and timer B bits of 02A1, since now the receiving edge for RS-232-C reception has been handled.
4. Reset the latches for timer B, DD06 and DD07, to \$FFFF.
5. Store the number of bits to send or receive plus one, held in 0298, into A8.

### NMI Interrupt Handler (VIC-20) FEAD-FF5B

#### Called by:

JMP at FEAA in NMI Interrupt Handler Jump.

On the VIC, NMI interrupts can be triggered by a high-to-low transition on the NMI line from pin W of the expansion port, , or from seven conditions on VIA #1. But only the following VIA #1 interrupt conditions are checked for in the NMI interrupt handler: CB1, timer 1, timer 2, RESTORE key, and autostart cartridge. To service other NMI sources from VIA #1, you must check these conditions yourself, possibly by modifying the RAM vector to execute your NMI routine before returning to the normal NMI interrupt handler in ROM.

The VIA #1 timer 1, timer 2, and CB1 interrupts are used for RS-232-C communications. Timer 1 interrupts control RS-232-C transmission. Timer 2 interrupts control reception of each bit for RS-232-C. CB1 interrupts detect when reception is to begin and initiates the input for each byte, setting initial values for timer 2, enabling the timer 2 interrupt, and disabling further CB1 interrupts while the byte is being received.

The VIC NMI interrupt handler allows nested NMI interrupts to occur. While this does not appear to prevent successful operation of RS-232-C I/O, it is more common for an NMI interrupt handler to service all possible causes of the NMI interrupt in one execution of the interrupt handler rather than to allow nested interrupts to occur.

## NMI Interrupts

### Operation:

1. Save the accumulator, X register, and Y register on the stack.
2. Test bit 7 of VIA #1 interrupt flag register, 911D, to see if the NMI interrupt occurred as a result of an interrupt on VIA #1. Branch to step 37 if bit 7 is 0 to restore the registers and RTI. Thus, exit if an NMI interrupt occurred from the expansion port. Remember that on the VIC-20 the RESTORE key connects to VIA #1.
3. Mask out all but the active interrupts by ANDing the interrupt flag register and the interrupt enable register. Save the result of 911D AND 911E in the X register.
4. Test for a RESTORE key interrupt, which appears as a CA1 interrupt, since the CA1 input line on VIA #1 comes from the RESTORE key. If this is not a RESTORE key interrupt, then branch to step 12.
5. If this is a RESTORE key interrupt, test for an autostart cartridge by JSR FD3F.
6. If an autostart cartridge is present, pass control to the cartridge warm start routine by JMP (A002).
7. Clear the RESTORE key interrupt using BIT 9111 (the port A VIA #1 data register).
8. JSR F734 to increment the jiffy clock and scan the keyboard. Thus, by continually hitting the RESTORE key on the VIC you can cause the jiffy clock to become incorrect since it should only be updated 60 times a second by the IRQ interrupt handler.
9. JSR FFE1 to see if the STOP key is down.
10. If the STOP key is not down, branch to step 12.
11. If STOP and RESTORE are down, fall through to the BRK interrupt handler routine which does the following:
  - JSR FD52 to initialize Kernal RAM vectors.
  - JSR FDF9 to initialize 6522 registers and enable VIA #2 timer 1 interrupts.
  - JSR E518 to initialize VIC chip registers, blank the screen, and set up the screen line link table.
  - JMP (C002) to BASIC's warm start.
12. LDA 911E, the VIA #1 interrupt enable register, ORA \$80, and save this value on the stack.
13. Store \$7F in 911E to disable all interrupts from VIA #1. This allows the IRQ output from VIA #1 to go high, preparing the 6502 to receive another NMI interrupt.

## NMI Interrupts

---

14. Reload accumulator from X register, which has the active interrupts from VIA #1.
15. Was this NMI interrupt caused by a VIA #1 timer 1 interrupt?
16. If not, branch to step 22.
17. If yes, then RS-232-C transmission is active. Set 911C, the peripheral handshaking control register to \$CE (1100 1110) and OR with B5, which contains the next RS-232-C bit to be transmitted in bit 5. Thus, a next bit of 1 causes the CB2 line to be held high (1), while a next bit of 0 causes the CB2 line to be held low (0). Since CB2 is the transmitted data line, the bit has now been transmitted.
18. Clear VIA #1 timer 1 interrupt flag by LDA 9114.
19. Pull the interrupt enable register (that was ORed with \$80) from the stack and store this value in 911E, thus re-enabling any interrupts that were active when the NMI interrupt occurred. Thus, the timer 1 interrupt is now re-enabled. (It's possible that timer 2 and CB1 interrupts are also now re-enabled.)
20. JSR EFA3 to the RS-232-C send routine.
21. JMP FF56 to restore the registers and RTI.
22. Reload accumulator with the cause of the interrupt.
23. Was the interrupt caused by a VIA #1 timer 2 interrupt?
24. If no, branch to step 31.
25. If yes, then RS-232-C reception of individual bits for a byte is active. LDA 9110, then AND \$01 to retrieve the value from PB0, the received data bit. This read of port B also clears the interrupt flag set by the previous CB1 interrupt.
26. Store this PB0 value into bit 0 of A7, the RS-232-C receiver input bit temporary storage location.
27. Reset timer 2 to a value based on the baud rate; clear timer 2 interrupt.
28. Restore accumulator with interrupt enable register (that was ORed with \$80) from the stack and store into the interrupt enable register, thus restoring the NMI interrupts that were enabled at entry. Timer 2 interrupts are now enabled.
29. JSR F036 to the RS-232-C receive routine.
30. JMP FF56 to restore registers and RTI.
31. Reload the accumulator with the cause of the NMI interrupt.

## NMI Interrupts

---

32. Was the interrupt a CB1 interrupt, the receive edge signal for RS-232-C? Note that opening an RS-232-C channel for input enables CB1 interrupts.
33. If not, then branch to step 37.
34. If yes, then a new byte is being received from the RS-232-C interface. Set timer 2 value from the baud rate table.
35. Enable timer 2 interrupts to allow timer 2 to control the sampling for the individual bits in this byte being received. Also disable CB1 interrupts.
36. Load number of bits to be sent or received from 0298 and store in A8.
37. Restore Y register, X register, and accumulator, then RTI.



## **Chapter 4**

---

# **IRQ Interrupts**





# IRQ Interrupts

---

The IRQ interrupt vector for the 6510/6502 is located at (FFFE). Whenever the microprocessor's IRQ input line is pulled low and IRQ interrupts are enabled, the address in (FFFE), which is FF48/FF72, is placed in the program counter. At FF48/FF72 a routine determines whether the interrupt was caused by an IRQ interrupt or by a BRK instruction. This routine then jumps to the address specified in either the IRQ interrupt handler vector at (0314) or in the BRK interrupt handler vector at (0316). The default addresses for these are EA31/EABF for the IRQ interrupt handler, and FE66/FED2 for the BRK interrupt handler. Since these vectors are in RAM, they can be changed. The tape I/O routines change the IRQ vector to FC6A/FCA8 for a tape header write, FBCD/FC0B for tape write, and F92C/F98E for tape read.

Normally, IRQ interrupts are produced every 1/60 second by timer A of CIA #1 (64) or timer 1 of VIA #2 (VIC). The frequency with which the IRQ interrupts occur can be modified by changing the CIA #1 timer A latches/VIA #2 timer 1 latches. However, by lowering the values in these latches, you do not increase the instruction execution speed of your 64 or VIC. You only increase the frequency with which IRQ interrupts are serviced. If you lower the timer latches too much, you might find yourself just servicing IRQ interrupts and not doing anything else.

The functions executed by the IRQ interrupt handler include updating the jiffy clock, testing for the STOP key, and scanning the keyboard.

## **IRQ/BRK Interrupt Switch** **FF48/FF72-FF5A/FF84**

### **Called by:**

The program counter loads the IRQ vector at (FFFE) when the IRQ input line is pulled low and IRQ interrupts are enabled.

Whenever a hardware- or software-generated IRQ interrupt occurs, the status register is pushed onto the stack. For a BRK instruction (software interrupt) the status register is first modified by setting the break flag (bit four of the status register) before it is pushed onto the stack. The routine then

## IRQ Interrupts

examines the status register that has been pushed onto the stack to determine whether this IRQ interrupt was caused by a BRK instruction or by a hardware IRQ interrupt. If the BRK flag is set, then the processor executes a JMP (0316) to the BRK interrupt handler. If not set, it performs a JMP (0314) to the IRQ interrupt handler. The default for the BRK handler vector at (316) is FE66/FED2. The default for the IRQ handler vector at (314) is EA31/EABF. This routine also saves the accumulator, X register, and Y register on the stack.

Let's examine in detail how this routine checks for the BRK flag in the status register on the stack. The 6510/6502 hardware automatically pushes the high order byte of the program counter, the low order byte of the program counter, and the status register onto the stack when an IRQ interrupt occurs. This routine then pushes the accumulator, X register, and Y register onto the stack.

Since the stack pointer points to the address in the stack that is one less than the bottom of the stack (the Y register is at the bottom of the stack), the status register is located four bytes above the location pointed to by the stack pointer. A TSX instruction transfers the stack pointer to the X register after all the registers are pushed onto the stack. Then, to retrieve the status register from the stack and place it into the accumulator, the routine executes a LDA \$0104,X (to load the byte that is four locations above the stack pointer). The stack, which builds downward, is located at 0100–01FF, with 01FF the initial top of the stack. The table below illustrates how this operation works using some arbitrary address locations on the stack.

### Saved Stack Contents on IRQ Interrupt

Stack Address	Stack Contents
---------------	----------------

0110	not yet used (stack pointer contains \$10; points here)
0111	Y register
0112	X register
0113	Accumulator
0114	Status register
0115	Program counter low byte
0116	Program counter high byte

TSX moves \$10 into X register

LDA \$0104,X loads accumulator with value of byte at \$0104

## IRQ Interrupts

+ X = \$0104 + \$10 = \$0114. At location 0114 you find the saved value for the status register.

### Operation:

1. Push accumulator, X register, and Y register onto the stack to allow restoration when the IRQ interrupt handler terminates. This allows the restoration of the register values of the program in execution when the IRQ interrupt is finished.
2. Load accumulator with the status register that has been pushed onto the stack by TSX and LDA \$0104,X.
3. See if the BRK flag was set in the saved status register.
4. If the BRK flag was set then JMP (0316) to the BRK interrupt handler at FE66/FED2.
5. If the BRK flag was not set then JMP (0314) to the IRQ interrupt handler at EA31/EABF.

### BRK Interrupt Handler FE66/FED2-FE71/FEDD

#### Called by:

Indirect JMP through (0316) at FF55/FF77 in IRQ/BRK Interrupt Switch; fall through from FE64/FED0 in NMI Interrupt Handler if STOP-RESTORE is pressed.

Whenever the STOP and RESTORE keys are held down (and no autostart cartridge is present), or when a BRK instruction executes, this routine is called.

This routine initializes the Kernal RAM vectors and 6526/6522 registers, enables CIA#1 timer 1/VIA#2 timer 1 interrupts, initializes the VIC-II/VIC chip registers, then jumps to BASIC's warm start.

#### Operation:

1. JSR FD15/FD52 to initialize the Kernal RAM vectors.
2. 64: JSR FDA3 to initialize the 6526 registers, 6510 I/O and data direction registers, 6581 SID chip registers, start CIA#1 timer A and enable timer A interrupts, and set serial clock line high.  
VIC: JSR FDF9 to initialize 6522 registers and enable VIA #2 timer 1 interrupts.
3. JSR E518 to initialize the VIC-II 6567/VIC 6560-6561 chip registers, clear the screen, and initialize the screen line link table.
4. JMP (A002)/(C002) to warm start BASIC. The warm start of BASIC begins at E37B/E467.

### IRQ Interrupt Handler EA31/EABF-EA86/EB1D

#### Called by:

Indirect JMP through (0314) at FF58/FF82 in IRQ/BRK Interrupt Switch.

The IRQ interrupt handler is entered whenever the system detects an IRQ interrupt. On the 64, timer A in CIA #1 generates IRQ interrupts, while on the VIC, timer 1 in VIA #2 generates them.

Numerous other sources can generate IRQ interrupts in addition to timer A/timer 1, but usually only the timer interrupts are enabled. This interrupt handler specifically clears only the interrupt flags for the timer.

The IRQ interrupt handler first updates the jiffy clock at A2-A0. It scans the STOP key column of the keyboard for any key value in this column and stores the value into the STOP key flag, 91.

Next, it checks to see if the cursor blink is enabled. If it isn't enabled, characters are in the keyboard queue, so skip the remainder of the operations described in the next paragraph.

If the cursor blink flag is enabled and it is time to blink the cursor, the routine determines whether the cursor blink flag indicates that the character under the cursor is reversed. If it is not reversed, the character under the cursor is in normal mode, so the program resets the cursor blink flag to indicate the character is reversed. The ASCII character under the cursor is saved at location CE, and the current color nybble is saved at 0287. The routine reverses the character under the cursor and displays this reversed character at the same screen location. When the character is in reverse mode, the opposite blinking action occurs.

The IRQ handler then checks to see if any tape buttons are down, if the tape motor interlock switch is set, and if timer A/timer 1 interrupts are enabled. If timer A/timer 1 interrupts are disabled, a tape button is down, and the tape interlock switch is zero, then the tape motor can be turned on. On the VIC 911C, the peripheral control register for VIA #1 must have the correct value to turn on the tape motor.

The routine scans the keyboard, and if a key is pressed it places its ASCII value in the keyboard buffer, clearing the interrupt flag for timer A/timer 1.

## IRQ Interrupts

Finally, it restores the Y register, the X register, and the accumulator from the stack and does an RTI.

### Operation:

1. JSR FFEA to the Kernal UDTIM routine. This routine increments the jiffy clock at A2-A0 and saves the scan value for the STOP key column in 91, the STOP key flag.
2. If the cursor blink switch at location CC is nonzero, branch to step 13 so that the cursor doesn't flash when characters are in the keyboard queue.
3. Decrement the countdown to the blink of the cursor, location CD, and if the result is not zero then branch to step 13.
4. Reset the blink countdown to \$14 if CD did reach zero.
5. Retrieve the character under the cursor and the color for this character. (D1) points to the start of the screen line the cursor is on, and D3 is the column of the cursor in the screen line.
6. If the cursor blink status flag at CF is 1, indicating reversed character under the cursor, then branch to step 11. The test for a value of 1 is to LSR CF and then BCS.
7. Set cursor blink status flag, CF, to 1 to indicate reversed character under cursor. Save the character under the cursor at CE.
8. JSR EA24/EAB2 (see chapter 7) to set pointer for color nybble to correspond to the start of the screen line.
9. Load the value of the color for the character under the cursor and save at 0287.
10. Retrieve the color of the nybble from 0286 and ASCII value of character from CE.
11. EOR the accumulator (which contains the ASCII value of the character) with \$80, flipping the high order bit. Each time this EOR is executed, the character is reversed since, by turning bit 7 on, characters from \$80-\$FF (the reverse character set) are displayed.
12. JSR EA1C/EAAA (see chapter 7) to store the character and the color on the screen.
13. See if any tape buttons (rewind, fast forward, play) are down. If so, branch to step 16.

The 64 test is LDA 01, AND \$10, BEQ EA71. Location 0001 bit 4 is the tape switch sense, and a value of 0 means a button is down.

## IRQ Interrupts

The VIC test is LDA 911F, AND \$40, BEQ EB01. Location 911F bit 6 is tape switch sense, and a value of 0 means a button is down.

14. If no buttons are down, set the tape motor interlock switch, C0, to zero. Since the IRQ interrupt occurs 60 times a second, C0 is continually reset to zero if no tape buttons are down.
15. Prepare to hold the output line to the tape motor high, and thus to turn off the tape motor if no buttons are down. On the 64, LDA 01, ORA \$20, and branch to step 19. On the VIC, LDA 911C, ORA \$02, and branch to step 18. On the VIC, the state of the CA2 line used for tape motor control is determined by bits 3–1 of location 911C. Bits 3 and 2 must both be 1 for the 1 in bit 1 to hold CA2 out high.
16. Come here from step 13 if a tape button is down. See if C0, the tape motor interlock, is zero. If not, branch to step 20.

Fall through to step 17 if a tape button is down and C0 is zero. Thus a nonzero C0 value prevents turning on the tape motor during the normal IRQ interrupt handler.

17. 64: LDA 01, AND \$1F (0001 1111 binary), thus forcing bit 5 tape motor control to 0.

VIC: LDA 911C, AND \$FD (1111 1101 binary), thus forcing bit 1 to zero, which will hold CA2 output low if bits 2 and 3 are both 1.

18. VIC only: If VIA #1 timer 1 interrupts (RS-232-C timing) are enabled, branch to step 20.
19. 64: STA 01. Either turn tape motor on or off.

VIC: STA 911C.

Step 19 should either turn on the tape motor (if entered after step 17 has been executed) or turn off the tape motor (if entered after step 15 has been executed). Because CA2 control is determined by three bits on the VIC, it is possible that the other two bits in 911C, bits 2 and 3, may have been modified and thus will not turn the tape on or off as expected.

20. JSR EA87/EB1E to do the keyboard scan.
21. 64: LDA DC0D to clear any CIA #1 interrupt flags and to allow the CIA #1 IRQ output to go high.

VIC: BIT 9124 to clear the timer 1 interrupt.

22. Restore Y register, X register, and accumulator from the stack.

23. RTI. Pull the status register and the low and high order bytes of the program counter from the stack.

## Jiffy Clock Update—STOP Key Scan F69B/F734-F6DC/F75F

### Called by:

JSR from Kernal UDTIM vector at FFEA (called by JSR at EA31/EABF during the IRQ Interrupt Handler), JSR at FECA in NMI Interrupt Handler (VIC only); alternate entry at F6BC by JSR at FE5E in NMI Interrupt Handler (64 only), JSR at F8CA in Reset IRQ Vector and Set Interrupt Enable Register (64 only).

The jiffy clock at A2-A0 is incremented. If the jiffy clock has reached a value equal to 24 hours, then the jiffy clock is reset to zero. The 64 NMI interrupt handler, unlike the VIC's, uses an alternate entry point into this routine which does not increment the jiffy clock.

Next, the keyboard column that contains the STOP key is scanned and the value of the keyboard row for the column is stored in the STOP key flag, location 91. If the STOP key is down, the value stored is \$7F (64), or \$FE (VIC). Also, in checking for the STOP key on the VIC, the CA1 interrupt flag is cleared.

### Exit conditions:

Jiffy clock incremented; reset to 0 if value reaches 24 hours. The possible values for the STOP key flag, 91, are shown below:

### STOP Key Flag Values

#### Value in

#### Location 91

#### (hex)

	64	VIC
FF	no key pressed	no key pressed
FE	1	STOP
FD	Back arrow	Left SHIFT
FB	CTRL	X
F7	2	V
EF	Space	N
DF	Commodore	,
BF	Q	/
7F	STOP	Cursor down

### Operation:

1. Increment the three-byte jiffy clock at A2-A0. If A0 rolls over from \$FF to \$00 then increment A1. If A1 rolls over from \$FF to \$00 then increment A2. As a result, with the value at A0 updated every 1/60 second (.01667 sec.), A1 is updated every 4.2667 seconds, and A2 every 18.2044 minutes.
2. Test to see if the jiffy clock has reached a value equal to 24 hours. If it has, all three bytes of the jiffy clock are reset to 0.
3. VIC: Load accumulator from VIA #2 port A data register at 912F, compare against itself to debounce the key, and then store the result in the STOP key flag at 91. If the STOP key was pressed, the value \$FE is stored.

*Steps 3-10 below apply only to the 64.*

3. F6BC: Load accumulator from DC01, CIA #1 data port B (keyboard row values) and compare against itself until equal to debounce (stabilize) the key.
4. Transfer accumulator to X register.
5. If the high order bit is on, then branch to step 10. The STOP key value in 91 is \$7F, and so it has its high order bit off. Thus, branch if the STOP key was not detected.
6. Load X register with \$BD and store in DC00, CIA #1 port A keyboard column output.
7. Load X register from DC01, CIA #1 port B (keyboard row) and compare against itself until equal to make it stabilize.
8. Store accumulator (which still has the same value as in step 4) in DC00, the CIA #1 port A keyboard column output.
9. Increment the X register. If not equal to zero, skip step 10 and branch to the RTS instruction to exit the routine. The X register would have been \$FF (and thus the INX would have made it zero) if no key was held down. If the STOP key (or any other key) was still held down, fall through to step 10.
10. Store accumulator in 91, the flag for the STOP key.



### Keyboard Scan

**EA87/EB1E-EB47/EBDB**

#### Called by:

JSR at EA7B/EB12 in IRQ Interrupt Handler, JMP from Kernal SCNKEY vector at FF9F.

This routine scans the keyboard to detect a keypress. If no key is pressed it exits. It finds which key has been pressed by scanning one column at a time, starting with column 0, until a column is found in which a key is being pressed. Each time a row is scanned, the Y register is incremented; the Y register is used as an index into whichever keyboard table is used. Once 64 keys have been scanned (eight rows by eight columns), a routine at EB48/EBDC sets up the keyboard table being used, based on the value in the shift control flag. This shift control flag is also updated in the keyboard scan routine if the key held down has a value of 1 (SHIFT), 2 (Commodore key), or 4 (CTRL).

The scan of rows and columns is completed only when all eight columns by eight rows have been scanned. Thus, a value could be stored for a SHIFT, Commodore, or CTRL key and for one of the other keys on the keyboard during one scan of the keyboard. Also, if you hold two keys down at the same time, only the key with the higher matrix value is detected.

Once the routine finds the matrix value for the key and selects the keyboard character set, it finds the ASCII value of the key.

Next, it tests to see if all keys, no keys, or normal keys repeat. SPACE, DELETE, INSERT, and the cursor keys are considered the normal repeat keys.

If the key is a repeat key and the delay before the initial repeat of a key has not yet reached zero, the program exits this routine. Once the initial repeat flag, 028C, is decremented to zero, then subsequent scans of the keyboard with the same key held down cause the flag for subsequent key repeats, 028B, to be decremented. When 028B reaches zero, the routine places the ASCII value of the key in the keyboard buffer, if the keyboard buffer contains one or no ASCII key values. If the buffer contains two or more ASCII key values, the program exits. 028B is reset to \$4 after each time it decrements to 0. 028C is reset to \$10 only when a different key is detected in this scan of the keyboard as compared to the last keyboard scan.

Thus, the initial repeat of a key takes longer than subsequent repeats because the initial repeat has to wait for 028C to decrement to zero while subsequent repeats don't have as long a delay.

If no keys are repeating, the routine saves the key just pressed as the value of the last key pressed, C5, and saves the shift pattern as the old shift pattern, 028E. Then, if the number of characters in the keyboard buffer is less than the maximum allowed, it stores the ASCII value of this key in the buffer and increments the number of characters in the buffer.

If the keyboard buffer is full, the ASCII key value is discarded. Finally, the program resets the column being scanned to column 7/column 3 to allow the scan for the STOP key, and exits.

The JMP (028F) at EADD/EB71, which calls the keyboard table setup at EB48/EBDC, returns from the keyboard table setup by way of a JMP EAE0/EB74 at EB76/EC43.

One programming trick is to set (F5), the pointer to the keyboard table being used, to an address where you have defined your own ASCII keyboard table. To do this, simply change the vector at (028F) to point to a routine where you load the address of your table into (F5).

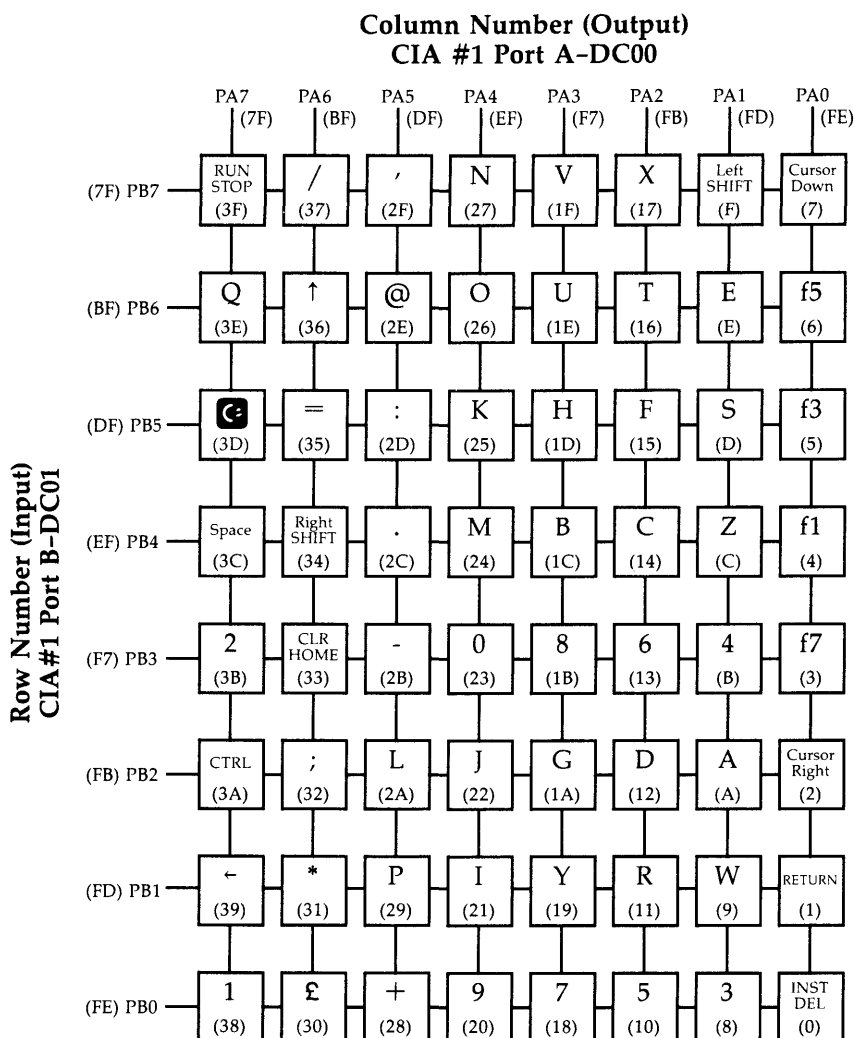
As mentioned, the organization of the 64 and VIC keyboards is based on a matrix of eight columns and eight rows. On the 64, the column number (output) is written to DC00 CIA #1 port A, and the row value read from DC01, CIA #1 port B. On the VIC, the column number (output) is written to 9120 VIA#2 port B, and the row value read from 9121 VIA#2 port A.

Reverse logic is used in detecting when a key is down in that *0 indicates a key is down*. The location where both the column is 0 and the row is 0 indicates which key in the matrix is pressed.

Figure 4-1 shows the keyboard matrix for the 64 and figure 4-2 shows the keyboard matrix for the VIC. Each key's matrix value is shown next to the key in parentheses. In the figures, PA7-PA0 correspond to bits 7-0 of data port A and PB7-PB0 correspond to bits 7-0 of data port B.

## IRQ Interrupts

**Figure 4-1. Commodore 64 Keyboard Matrix**



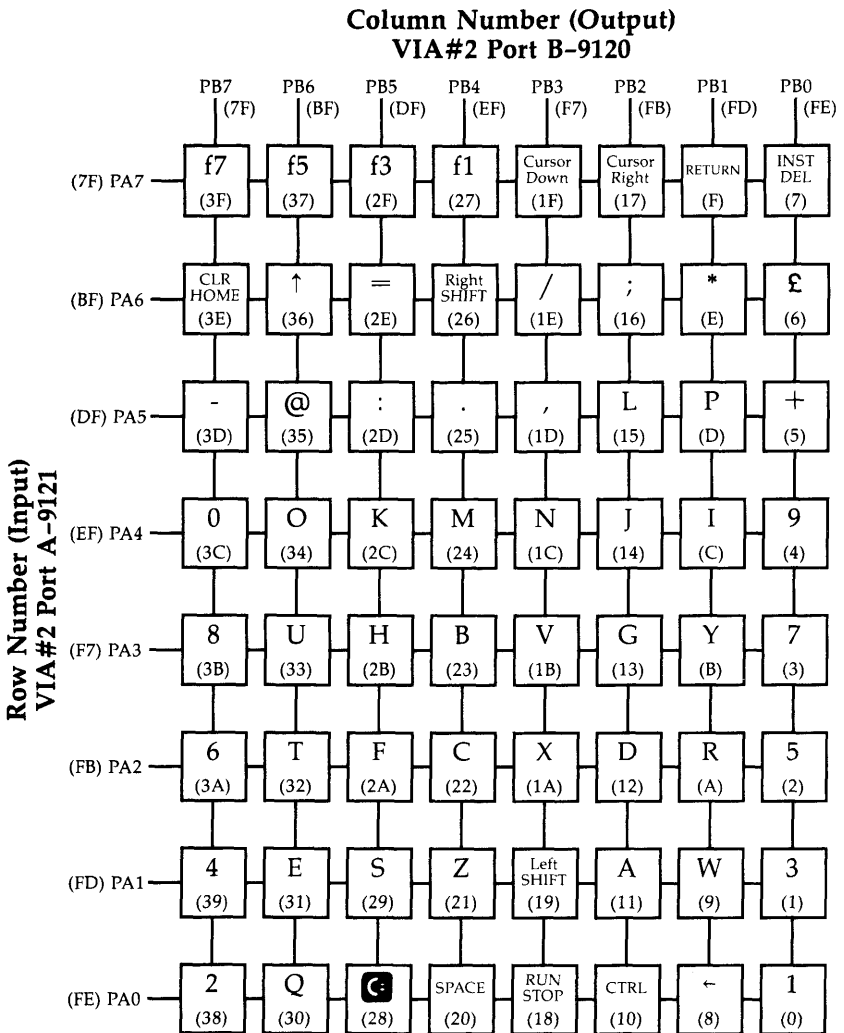
The number in parentheses below each key is that key's matrix value (in hex).

A value of \$FF in DC01 means no key pressed.

The matrix pictured here is for unshifted keys.

## IRQ Interrupts

**Figure 4-2. VIC-20 Keyboard Matrix.**



The number in parentheses below each key is that key's matrix value (in hex).

A value of \$FF in 9121 means no key pressed.

The matrix pictured here is for unshifted keys.

## IRQ Interrupts

Throughout most of the 64/VIC keyboard reading routines, only one column is brought low at a time, while the rest remain high. This column that is brought low is reset by all routines that modify the column to \$7F/\$F7 before the exiting. This restored column value is the column used to scan for the STOP key.

### Operation:

1. Initialize the shift control flag, 028D, to zero to indicate that the SHIFT, Commodore, or CTRL key is not pressed.
2. Initialize the current key pressed flag, CB, to \$40 to indicate none pressed.
3. 64: Store \$00 in DC00, CIA #1 data port A, the keyboard column. VIC: Store \$00 in 9120, VIA #2 data port B. This brings all output columns low.
4. Read the keyboard row, DC01/9121. If the value read is \$FF, no key in the matrix has been pressed. Branch to step 51 if no key is pressed to save the last key pressed, the last shift pattern, and exit.
5. Store \$FE in DC00/9120 to bring column 0 low to start the scan of individual keyboard columns.
6. Initialize the index into the ASCII keyboard table (the Y register) to zero.
7. Set a default ASCII keyboard table of EB81/EC5E (upper case normal) in (F5).
8. Set the number of times to shift the keyboard row values to eight, using the X register for this count. On the 64, also PHA, saving the current column to scan on the stack.
9. Read the keyboard row port DC01/9121. Stabilize (debounce) the key by comparing the current and last values read from the port until these values are equal.
10. LSR to shift the keyboard row being examined into the carry bit.
11. If this LSR sets the carry, then a 1 was shifted in, which means that particular row was not the row on which a key was pressed. BCS to step 20.
12. If the carry was clear, that row was the one on which the key was pressed. The column for this key is also a zero. The matrix counter indicates which key is pressed.
13. PHA, pushing the current row status onto the stack.
14. LDA (F5),Y to retrieve the value of the key pressed from the normal character set.

## IRQ Interrupts

15. Is this key's ASCII value  $\geq 5$  or  $= 3$ ? If it is, then branch to step 18.
16. If not, then its value must be either 1, 2, or 4. A value of 1 indicates the SHIFT key, 2 is the Commodore key, and 4 is the CTRL key. ORA this value with the shift control flag at 028D.
17. Branch to step 19.
18. Store the current value of the Y register at CB, the matrix coordinate of the current key pressed (the index into the keyboard table). Note that this is when a row was found with a 0, and the column was also 0.
19. PLA to restore the row inspection status.
20. Increment the Y register, the index into the keyboard character set.
21. Is Y register  $\geq \$41$  (65 decimal)? If it is, all eight rows by eight columns have been scanned.
22. If all rows and columns have been scanned, branch to step 25.
23. If the entire keyboard has not been scanned, decrement the X register, which is the number of times to shift the keyboard row.
24. If the number of times to shift the row is not equal to zero, branch to step 10 to examine the next row of key values.
25. If the number of times to shift the row is zero, set the carry in preparation for a ROL.
26. 64: PLA then ROL and STA DC00.  
VIC: ROL 9120.  
This rotates the keyboard column left through the carry. By setting the carry before the ROL, a 1 is rotated into bit 0, thus leaving the keyboard column still containing just one bit that is zero.
27. BNE to step 8. This branches as long as Z flag is clear. The Z flag is set to 1 when a zero is rotated back into bit 0, indicating all columns have been scanned.
28. JMP (028F) with a default of EB48/EBDC to the keyboard table setup routine.
29. The keyboard setup routine has a common exit that jumps back to the instruction immediately following the JMP (028F) in step 28.
30. Load the Y register index into the character set from CB, the matrix key value.

## IRQ Interrupts

31. Load the ASCII value of the key pressed, pointed to by (F5),Y.
32. Was this Y index the same as the last time? C5 holds the last matrix value. If yes, then branch to step 34.
33. If not, then set the delay before the key repeat of \$10 in 028C and branch to step 51.
34. Steps 34–50 handle the key repeat logic.  
    AND \$7F with accumulator, which contains the ASCII value of the key pressed.
35. BIT 028A, the key repeat flag. 028A holds \$80 if all keys repeat, \$40 if none repeat, \$00 for normal repeat keys.
36. If all keys repeat, BMI to step 41.
37. If no keys repeat, BVS to step 63.
38. If normal keys repeat then fall through to step 39.
39. If the key value was \$FF—which after the AND in step 34 is now \$7F—meaning no key down, then branch to step 51.
40. If the key is not the space key, one of the cursor keys, the INSerT key, or the DELeTe key, then branch to step 63.
41. LDY 028C, the delay before the initial repeat of a key.
42. If zero, branch to step 45.
43. If not zero, delay = delay – 1 (DEC 028C).
44. Is delay, 028C, now zero? If no, branch to step 63.
45. If yes, then decrement 028B, the delay before following repeats if the key is still held down.
46. If 028B is now zero, continue; if not, branch to step 63.
47. Reset the delay before following repeats of the key, 028B, to 4. Second and subsequent repeats of a key are faster than the first repeat because on subsequent repeats 028C has already counted down to 0.
48. LDY with the number of characters in the keyboard buffer, C6.
49. DEY.
50. If Y register is  $\geq 0$ , indicating the keyboard buffer contains two or more key values, branch to step 63.
51. LDY with the index of the key just pressed from CB.
52. STY C5, the index of the key last pressed.
53. Load the current shift pattern, 028D.
54. Save as the last shift pattern, 028E.
55. Was the value of the key \$FF (no key pressed)?
56. If yes, then branch to step 63.

57. Restore accumulator with ASCII key value saved in X register.
58. Load number of characters in keyboard buffer from C6.
59. Is the number of character  $\geq$  the maximum number allowed in 0289?
60. If yes, branch to step 63. In effect, discard all keys typed as long as the keyboard buffer is full.
61. Save the ASCII value of the key pressed in the keyboard buffer at the location one byte past the last ASCII value in the buffer.
62. Increment C6, the number of characters in the keyboard buffer.
63. Set DC00 to \$7F/9120 to \$F7, thus setting the column value for scanning the STOP key.

### Keyboard Table Setup EB48/EBDC-EB78/EC45

#### Called by:

Indirect JMP through (028F) at EADD/EB71 in Keyboard Scan.

The value in the shift control flag, 028D, determines which keyboard character set is used. If this flag is 3, both the SHIFT and Commodore keys were held down. If the current shift pattern in this scan of the keyboard, 028D, is the same as the previous pattern in the SHIFT keys, 028E, the routine ends, leaving the keyboard table unchanged.

If the shift patterns are different and the SHIFT keys are enabled (bit 7 of 0291 = 0), EOR bit 1 of D018/9005, flipping the character set between upper- or lowercase character sets.

If the value in 028D was not 3, shift the value left one bit (ASL). Entry from the keyboard scan routine makes the only permissible values at entry in 028D to be 0, 1, 2, 3, 4, 5, 6, or 7 (the values for SHIFT, Commodore, or CTRL keys, or combinations thereof). The ASL thus results in a value of 0, 2 (SHIFT), 4 (Commodore key), 8 (CTRL key), or a value greater than 8 if a combination of keys (remember that the original value of 3 was already checked for).

Compare to see if after the ASL the value is less than 8. If so, that value is used as an index into the keyboard table character set select. However, if the value is  $\geq$  8, then the index is reset to 6 (the index for the CTRL key character set).



Thus, 6 is the index for the CTRL character set, 4 is the index for the Commodore key character set, 2 is the index for the SHIFT character set, and 0 is the index for normal character set.

This routine exits by jumping back into the keyboard scan routine. The VIC version of this routine contains numerous NOP instructions and extra unused keyboard table values and an extra unused keyboard character set. The 64 version is considerably cleaned up.

### Operation:

1. See if Commodore and SHIFT keys are both down.
2. If no, branch to step 7.
3. If yes, see if the last shift pattern, 028E, is the same as this shift pattern.
4. If the patterns are the same, then exit. If the patterns are different, see if SHIFT keys are enabled, which is indicated by a 0 in bit 7 of 0291.
5. If SHIFT keys are disabled, exit.
6. If SHIFT keys are enabled, then flip the current character set, whether it is normal or reversed, to uppercase/graphics if it is lowercase/uppercase, and to lowercase/uppercase if it is uppercase/graphics; exit.
7. SHIFT key flag, 028D, is in the accumulator now. ASL this SHIFT key value.
8. Is accumulator < 8. If yes, branch to step 10.
9. If no, then reset accumulator to 6.
10. TAX to set X register to keyboard table index.
11. Use X as index into character set using table at EB79/EC46, and save address of the table in (F5).



## **Chapter 5**

---

# **Kernal Routines**



# Kernal Routines

---

A section of the Kernal from FF81/FF8A through FFF5 contains JMP instructions for the 39/36 routines that have been defined by Commodore as the user-callable Kernal routines. Note that there are three more for the 64 than for the VIC.

These JMP instructions are intended to allow you to write programs that use any of these Kernal functions without having to wonder if they will still work on later Commodore computers or if the Kernal ROM is modified on the 64 or VIC. Thus, if a routine moves to a different memory location, you need not be concerned if you just JSR to the Kernal entry which is in the JMP instruction table.

Another reason to use the standard jump instruction table is that you don't have to be concerned with the internal workings of the routines. Rather, you just provide the information that is needed by the routine. The routines execute the function you request, returning information if that is part of the function and providing a means of error detection, either through the carry flag and the accumulator or the status byte, location 90.

Since these jump instructions are a standard feature of Commodore Kernal ROMs (at least on the VIC and 64), you may wonder why anyone would not use the jump instructions. Some reasons follow. If the jump vector does an absolute JMP (such as at FFB1, where there is a JMP ED0C/EE17 for the send-listen-with-attention-to-serial-device function), you cannot modify this serial function if you use the standard jump entry.

If you want to provide an additional feature that precedes or follows the standard routine, you may find using the Kernal jump instructions awkward.

For the jump table entries that jump indirectly, such as CHRIN, which does a JMP (0324), you can just change the vector at 0324 to point to your own routine. However, not all the jump instructions use indirect JMPs. See the table below for details of which use indirect jumps and which use absolute jumps. Perhaps the reason Commodore doesn't use indirect jumps for all the vectors is the lack of free space in pages 0, 2, and 3.

## Kernal Routines

---

A jump vector that appears unused is the SETTMO routine. Also, the RAM vector at (032E) is not called.

The following list compares the Kernal RAM vector table starting at 031A to the way this function is called from the Kernal jump instruction table.

### Kernal RAM Vectors and Kernal Jump Table Usage

Vector	Function	Jump Instruction
(031A)	OPEN	JMP (031A)
(031C)	CLOSE	JMP (031C)
(031E)	CHKIN	JMP (031E)
(0320)	CHKOUT	JMP (0320)
(0322)	CLRCHN	JMP (0322)
(0324)	CHRIN	JMP (0324)
(0326)	CHROUT	JMP (0326)
(0328)	STOP	JMP (0328)
(032A)	GETIN	JMP (032A)
(032C)	CLALL	JMP (032C)
(032E)	USRCMD	Not called
(0330)	LOAD	JMP F49E/F542
(0332)	SAVE	JMP F5DD/F675

This chart shows that only LOAD and SAVE do not execute a JMP indirect through a vector upon entering the Kernal table. Both LOAD and SAVE jump to routines which perform some setup operations before doing the JMP indirect through their vectors. LOAD sets the vector at (C3) as the starting address for the LOAD from the X and Y registers before doing a JMP (0330). SAVE sets (AE) to the end of the save area +1 from the X and Y registers, and (C1) to the start of the save area from a page 0 location whose address is in the accumulator, then SAVE does a JMP (0332). If you modify LOAD or SAVE, be aware of this preparation. There seems to be little reason why Commodore doesn't just do a jump indirect for SAVE and LOAD as they do for the other functions that have a RAM vector. If you are writing your own LOAD or SAVE, and you're still doing the JMP FFD5 for LOAD and JMP FFD8 for SAVE, you may not want (C3), (AE), or (C1) modified before they get control.

Another reason for not using the Kernal jump instruction is if it points to a routine that functions incorrectly. For example, the Kernal jump instructions for CHROUT and CHRIN that can be used for RS-232 operations do not function correctly for RS-232 x-line handshaking on the VIC.

## Kernal Routines

A third reason not to use the jump instruction tables is if you want to use a section of Kernal ROM that doesn't have a Kernal table entry. For example, you can create autoload/autostart machine language tapes, but only by directly calling the Kernal ROM routines, since there are no JMP vectors to the necessary routines. Another example is if you want to use the screen editor routines from your machine language program, which are not available through the Kernal jump vectors.

In summary, the Kernal jump instructions do have a purpose, and you should use them when appropriate. However, the jump instructions cannot or should not always be used, and after reading the rest of this book you may find Kernal ROM routines that you want to use directly without using the jump instructions.

Commodore intends the jump instructions to provide you with a bridge to a different version of a Kernal ROM without your having to rewrite your machine language program. While this jump instruction compatibility is true for VIC-to-64 translations, it is not true for 64-to-VIC translations because there are three new 64 jump instructions that are not provided in the VIC.

The alphabetical table of jump instructions below should be useful when you are programming. The *Operand* column lists the place to which the JMP transfers control, either an absolute location or a location pointed to by an indirect vector. *Default* gives the address found in the RAM indirect vector if it has not been modified.

### Kernal Jump Table

Name	Address	Operand	Default	Description
ACPTR	FFA5	EE13/EF19		Get a byte from serial bus
CHKIN	FFC6	(031E)	F203/F2C7	Open channel for input
CHKOUT	FFC9	(0320)	F250/F309	Open channel for output
CHRIN	FFCF	(0324)	F157/F20E	Get a byte from input channel
CHROUT	FFD2	(0326)	F1CA/F27A	Send a byte to output channel
CIOUT	FFA8	EDDD/EEEE		Send a byte to serial bus
CINT	FF81	FF5B (64 only)		Initialize screen editor
CLALL	FFE7	(032C)	F32F/F3EF	Close all channels and files
CLOSE	FFC3	(031C)	F291/F34A	Close logical file
CLRCH	FFCC	(0322)	F333/F3F3	Reset I/O channels
GETIN	FFE4	(032A)	F13E/F1F5	Retrieve character from channel
IOBASE	FFF3	E500		Return base address of I/O registers
IOINIT	FF84	FDA3 (64 only)		Initialize I/O devices
LISTEN	FFB1	ED0C/EE17		Send listen with attention to serial devices

## Kernal Routines

---

Name	Address	Operand	Default	Description
LOAD	FFD5	F49E/F542		LOAD/VERIFY to RAM
MEMBOT	FF9C	FE34/FE82		Read or set the start-of-memory pointer
MEMTOP	FF99	FE25/FE73		Read or set the end-of-memory pointer
OPEN	FFC0	(031A)	F34A/F40A	Open logical file
PLOT	FFF0	E50A		Read or set cursor location
RAMTAS	FF87	FD50 (64 only)		Memory initialization
RDTIM	FFDE	F6DD/F760		Read jiffy clock into registers
READST	FFB7	FE07/FE57		Read or reset status
RESTOR	FF8A	FD15/FD52		Reset RAM vectors to default
SAVE	FFD8	F5DD/F675		Save contents of memory to device
SCNKEY	FF9F	EA87/EB1E		Detect keyboard entry
SCREEN	FFED	E505		Return number of columns and rows
SECOND	FF93	EDB9/EEC0		Send secondary address after listen to serial
SETLFS	FFBA	FED0/FE50		Set logical file number, device number, and secondary address
SETMSG	FF90	FE18/FE66		Set message control
SETNAM	FFBD	FDF9/FE49		Establish filename
SETTIM	FFDB	F6E4/F767		Set jiffy clock from registers
SETTMO	FFA2	FE21/FE6F		Set IEEE time-out
STOP	FFE1	(0328)	F6ED/F770	Test for STOP key
TALK	FFB4	ED09/EE14		Send talk with attention to serial devices
TKSA	FF96	EDC7/EECE		Send secondary address after talk to serial
UDTIM	FFEA	F69B/F734		Increment jiffy clock
UNLSN	FFAE	EDFE/EF04		Send unlisten to serial
UNTLK	FFAB	EDEF/EEF6		Send untalk to serial
VECTOR	FF8D	FD1A/FD57		Read or set RAM vectors

The following sections discuss the above jump instructions. Some routines used by these jump instructions are included here, if they are not discussed elsewhere. For example, the LOAD and SAVE routines that are jumped to are discussed here, since these routines can be used for both serial devices and tape. The specific parts of LOAD and SAVE that apply to serial or to tape are discussed in those sections. Routines, such as those jumped to by TALK, SECOND, or TKSA, that are specific to one device or topic are discussed in those sections.

### ACPTR FFA5

**Called by:**  
None.



### **Setup routines:**

TALK, TKSA

The vector is JMP EE13/EF19. At EE13/EF19 the computer goes through a handshake sequence with the serial bus. During this sequence, the EOI handshake is performed if the serial clock input line does not go low within 250 microseconds as expected. If the EOI handshake sequence is performed, the routine sets the EOI status in the I/O status word, location 90. It can set the time-out status in the status word if serial clock in fails to go low within a certain time range.

If the preparation to receive handshaking signals detects no problems, and if the eight bits are received without handshaking error, the routine returns the byte received in the accumulator.

### **Exit conditions:**

The accumulator contains the byte received from the serial bus.

## **CHKIN**

**FFC6**

### **Called by:**

JSR at E11E/E11B in BASIC's Set Input Device.

### **Setup routines:**

OPEN

### **Entry Requirements:**

The X register should contain the logical file number.

JMP (031E) with a default of F20E/F2C7. If the logical file is in the logical file number table, the routine obtains the device number and secondary address for this logical file from the corresponding entries in the device number and secondary address tables. If the logical file is not in the logical file number table, it displays FILE NOT OPEN, and returns with carry set and accumulator set to 3.

If the current device is the screen or the keyboard, the routine stores 0 for the keyboard or 3 for the screen in 99, the location holding the device number of the current input device. You don't have to use OPEN and CHRIN to input from the keyboard.

If the current device is the tape, the routine also checks the secondary address. If the current secondary address is not

\$60, the routine displays the NOT INPUT FILE message, and returns with carry set and accumulator set to 6. If the current secondary address is \$60, then location 99 is set to 1 to make tape the current input device. OPEN does an ORA \$60 of the secondary address.

If the current device is a serial device, it opens the input channel by sending a TALK command to the device, and sending the secondary address if the value for secondary address held in B9 is < 128 (decimal). If the serial device does not respond, it displays the DEVICE NOT PRESENT error message and returns with carry set and accumulator set to 5. Otherwise, it stores the serial device number in 99.

If the current device is RS-232, the routine opens an RS-232 input channel. This RS-232 routine sets the current input device, location 99, to 2 for RS-232, then handles either the 3-line handshaking or the x-line handshaking opening sequence.

### **CHKIN Execution** **F20E/F2C7-F236/F2EF**

#### **Called by:**

Indirect JMP through (031E) from Kernal CHKIN vector at FFC6.

If the current logical file passed in the X register is in the logical file number table, obtain its corresponding device number and secondary address from the device number and secondary address tables. If it is not in the logical file number table, exit with FILE NOT OPEN error message.

If the device is the screen or the keyboard, set location 99, the current input device number, from BA, the current device number, and exit.

If the current device is an RS-232 device, JMP to the Open RS-232 Device routine.

If the current device is a serial device, JMP to the Open Serial Input Channel routine.

If the current device is tape, see if the secondary address indicates reading from tape. If not, JMP to display the NOT INPUT FILE message.

Store the current device number in the input device number, 99, CLC, and exit.

### Operation:

1. JSR F30F/F3CF to see if the logical file number in the X register exists. If the logical file passed in the X register is not in the logical file number table, JMP F701/F784 to FILE NOT OPEN error message, set accumulator to 3, set the carry, and exit.
2. JSR F31F/F3DF to set the current logical file number in B8, the current device number in BA, and the current secondary address in B9 from the tables for the logical file, device number, and secondary address.
3. If the current device, BA, is the keyboard (0), or the screen (3), branch to step 8.
4. If the current device number is > 3, the current device is a serial device; branch to F237/F2F0 to open a logical file for a serial device.
5. If the current device is an RS-232 device, JMP F04D/F116 to open an RS-232 logical file as an input channel.
6. If the current device is tape, see if the secondary address is \$60. If the secondary address is \$60, branch to step 8. The secondary address of \$60 is set during the OPEN Execution routine when the secondary address is ORed with \$60.
7. If the secondary address is not \$60, JMP F70A/F78D to display the NOT INPUT FILE message and exit with the accumulator set to 6 and the carry set.
8. STA (the current device number is in the accumulator) into 99, the input device number.
9. CLC and RTS.

### CHKOUT FFC9

#### Called by:

JSR at E4AE/E115 in BASIC's Set Output Device.

#### Entry requirements:

Set X register to logical file number.

JMP (0320) with default of F250/F309. If the logical file is in the logical file number table, obtain the device number and secondary address for this logical file from the corresponding entries in the device number and secondary address tables. If the logical file is not in the logical file number table, display the FILE NOT OPEN message, and return with carry set and accumulator set to 3.

If the current device is the keyboard, display the NOT OUTPUT FILE message, and return with carry set and accumulator set to 7.

If the current device is the screen, just set 9A, the current output device, to 3, and exit. You do not have to call OPEN and CHROUT to display on the screen.

If the current device is tape, also check the secondary address. If the secondary address is not \$61, display the NOT OUTPUT FILE message, and return with carry set and accumulator set to 7. If the current secondary address is \$61, set 9A to 1 for tape. Note: OPEN does an ORA \$60 of the secondary address.

If the current device is a serial device, open the output channel for a serial device. Do this by commanding the current device to listen. Then for secondary addresses < 128, set the serial attention output line high. If the serial device does not handshake as expected, display DEVICE NOT PRESENT, and return with carry set and accumulator set to 5. Otherwise, set 9A to the serial device number.

If the current device is RS-232, then open an RS-232 output channel. This routine sets 9A to 2, and then it handles the 3-line or x-line handshaking sequence.

### **CHKOUT Execution** **F250/F309-F278/F331**

#### **Called by:**

Indirect JMP through (0320) from Kernal CHKOUT vector at FFC9.

If the logical file number passed in the accumulator at entry is not in the logical file table, display the FILE NOT OPEN error message.

If the logical file is in the file number table, obtain the current device number and secondary address for this logical file.

If the device is the keyboard, display the NOT OUTPUT FILE error message.

If the device is the screen, store the device number in the output device number, 9A, and exit.

If the device is a serial device, branch to Open Serial Output Channel.

If the device number is 2 (RS-232), jump to Open RS-232 Output Channel.

If the device is tape, the secondary address must not be \$60 because this indicates read from tape. If \$60 is found, display the NOT OUTPUT FILE error message. If the secondary address is legal, set the output device number, 9A, to the value 1.

### Operation:

1. JSR F30F/F3CF to see if the logical file number passed in the X register is in the logical file number table. If not, JMP F701/F784 to display the FILE NOT OPEN error message and return with 3 in accumulator and carry set, then exit.
2. JSR F31F/F3DF to obtain the current device number and the current secondary address from their respective tables.
3. If the current device is the keyboard, JMP F70D/F790 to display the NOT OUTPUT FILE error message, set accumulator to 7, set carry, and exit.
4. If the current device is the screen, store the device number in 9A, the output device number, then CLC, and exit.
5. If the current device is a serial device, branch to F279/F332 to Open Serial Output Channel.
6. If the current device is an RS-232 device, JMP EFE1/F0BC to Open RS-232 Output Channel.
7. If the current device is tape, the secondary address must not be \$60 (read tape). If the secondary address is \$60, JMP to NOT OUTPUT FILE error, set accumulator to 7, set carry, and exit. If the secondary address is legal, set the output device number, 9A, to 1 (tape).
8. CLC and RTS.

### CHRIN FFCF

#### Called by:

JSR at E112/E10F in BASIC's Input a Character.

#### Setup routines:

OPEN, CHKIN (not required in retrieving from keyboard).  
JMP (0324) with default of F157/F20E.

If the current input device, 99, is tape, then return the next byte from the tape buffer. Also, read one byte ahead to

see if the next byte is zero, indicating end of file, and if true, set end-of-file status in 90.

If the current input device, 99, is a serial device, the accumulator returns the byte received over the serial bus. However, if there are any I/O status errors, return with accumulator set to \$0D

If the current input device, 99, is RS-232, return with the next character from the RS-232 receive buffer. However, if the receive buffer is empty, the RS-232 routine on the VIC just loops until the receive buffer contains a character. The VIC can hang in an infinite loop if the RS-232 receive buffer never gets another character. If the receive buffer is empty on the 64, the routine returns with \$0D in the accumulator.

If the current input device is the keyboard, each character typed (except for control characters such as the cursor keys) is displayed on the screen until the unshifted RETURN is entered. Once an unshifted RETURN is typed, reset the input routine to retrieve a character from this screen line. After each character is retrieved from the screen line, increment the pointer to the character being retrieved in this logical line. The screen POKE code is converted to the equivalent ASCII code, which is returned in the accumulator. If the end of the screen line has been reached, then return \$0D, the ASCII code for a carriage return. The screen editor routines limit the size of a logical line to 80/88 characters. The way this CHRIN from the keyboard is typically used is to fill a buffer as BASIC does. BASIC calls the CHRIN routine to fill the BASIC input buffer at 0200. The BASIC routine keeps putting characters in the buffer until CHRIN retrieves a carriage return (ASCII \$0D).

If the current input device, 99, is the screen, then return the ASCII code for the screen character in the current logical line pointed to by D3, the column the cursor is on. D3 is then incremented to point to the next character in the line. If D3 has reached the end of the line, return \$0D signifying carriage return, and set D0 to 0 to force the next CHRIN to come from the keyboard.

When doing CHRIN from the keyboard, the keyboard routine uses this CHRIN from the screen once the carriage return has been entered. After processing the screen characters, the screen CHRIN then resets a flag at D0 to 0 to force input from the keyboard for the next CHRIN.

**Exit conditions:**

Accumulator holds byte returned from channel.

**Determine Input Device  
F157/F20E-F178/F22F****Called by:**

Indirect JMP through (0324) from Kernal CHRIN vector at FFCF.

Call the appropriate character input routine based on the input device number, 99.

**Operation:**

1. If 99 is set to 0 (keyboard), save D3 in CA, save D6 in C9, and JMP E632/E64F (see chapter 7) to receive a character from the keyboard.
2. If 99 is set to 3 (screen), store 3 in D0, save D5 in C8, and JMP E632/E64F (see chapter 7) to receive a character from the screen.
3. If 99 is set to 2 (RS-232), branch to F1B8/F26F (see chapter 9) to receive a character from the RS-232 device.
4. If the value in 99 > 3 (serial), branch to F1AD/F264 (see chapter 8) to receive a character from the serial device.
5. If 99 is set to 1 (tape), fall through to F179/F230 to receive a character from tape.

**CHROUT  
FFD2****Called by:**

JSR at E10C/E109 in BASIC's Output a Character, JSR at F135/F1EC in Display Kernal Message, JSR at F5C9/F661 in Display Filename, JSR at F726/F7A9 in Error Message Handler, JSR at F759/F7DC in Find Next Tape Header.

**Setup routines:**

OPEN, CHKOUT (not required if output device is the screen).

**Entry requirements:**

Accumulator should contain the character to be output, in CBM ASCII. JMP (0326) with a default of F1CA/F27A.

If 9A, the current output device, is the screen (3), the ASCII code is displayed on the screen unless the ASCII code is a screen control function (cursor key, DELeTe, INSeRT, and so

on). If the character is a control code, the routine performs the action. If the ASCII code is a valid screen display code, the code is displayed on the screen at the current cursor position and then the cursor is advanced to the next position on the screen.

If the current output device, 9A, is a serial device, (> 3), then JMP EDDD/EEE4 to send the character to all open serial devices. When sending a character to a serial device, a one-byte buffer, 95, is maintained. If this buffer is empty, the character to be output is simply stored in the buffer. If the buffer already contains a character, the routine sends the character from the buffer onto the serial bus and stores the character to be output in the buffer. When the serial file is closed or the serial device is commanded to unlisten, the final byte in the buffer is sent.

If the current output device, 9A, is RS-232 (2), the character to be output is stored in the RS-232 transmit buffer, and transmission is started if this is the first byte to be sent.

If the current output device, 9A, is tape (1), store the character in the currently available position in the tape buffer and increment the index to the available position in the tape buffer. Once the index is set to 192, write the tape buffer to tape. Then set the first byte of the tape buffer to 2 (identification for a data buffer) and reset the index to point to the second byte of the tape buffer.

Although the character to be output is in ASCII code for output to the screen, this is not the case for RS-232, serial, or tape. For example, if you are storing bytes to tape containing a code other than ASCII, CHROUT will send them to the tape buffer. For the screen, though, the 64/VIC screen editor is set up to convert ASCII codes to screen codes or screen functions, and would not function well if you did not use ASCII.

### **Determine Output Device** **F1CA/F27A-F1E4/F28E**

#### **Called by:**

Indirect JMP through (0326) from Kernal CHROUT vector at FFD2.

Call the appropriate character output routine based on the output device number, 9A.



### Operation:

1. If the output device is the screen, JMP E716/E742 (see chapter 7) to output a character to the screen.
2. If the output device is a serial device, JMP EDDD/EEE4 (see chapter 8) to output a character to the serial device.
3. If the output device is an RS-232 device, branch to F208/F2B9 (see chapter 9) to output a character to an RS-232 device.
4. If the output device is tape, fall through to F1E5/F28F (see chapter 10) to handle CHROUT to tape.

### CINT (64 only)

**FF81**

#### Called by:

None.

JMP FF5B to initialize the VIC-II (6567) chip registers, clear the screen, set the cursor pointer, initialize the screen line link table, set the PAL/NTSC flag, set value for CIA #1 timer A, enable interrupts for CIA #1 timer A, and start timer A.

This Kernal jump instruction is only available on the 64. The nearest equivalent on the VIC is to JSR E518 to set the VIC (6560–6561) chip registers, clear the screen, set the cursor pointers, and initialize the screen line link table. The 6560 and 6561 chips are, respectively, the NTSC and PAL versions of the VIC's video chip. The VIC equivalent of CINT for enabling the IRQ timer interrupt is to JMP FE39 to enable VIA #2 timer 1 interrupts and to set a timer 1 value.

Thus, a VIC version might be something like this:

```
.... JMP 02A1
02A1 JSR E518
02A4 JSR FE39
02A7 RTS
```

CINT, or its VIC equivalent, is only needed if you write an autostart cartridge program and need to use the screen editor or IRQ timer A/timer 1 interrupts. If no autostart cartridge exists, the 64/VIC performs the actions in CINT during system reset.

## **CIOUT FFA8**

**Called by:**  
None.

**Setup routines:**  
LISTEN, SECOND (if serial device requires a secondary address)

**Entry requirements:**  
Accumulator should contain character to output. JMP EDDD/EEE4 to execute the Send Serial Byte Deferred routine.

When sending a character to a serial device, the routine maintains a one byte buffer at 95. If this buffer is empty, the character to be output is simply stored in the buffer. If the buffer already contains a character, the character from the buffer is sent onto the serial bus and the character to be output is stored in the buffer. When the serial file is closed or the serial device commanded to unlisten, the final byte in the buffer is sent. The character is sent to all open devices on the serial bus.

## **CLALL FFE7**

**Called by:**  
JSR at A660/C660 in BASIC's CLR.  
JMP (0322) with a default of F32F/F3EF.  
Set 98, the number of currently open files, to 0.  
If the current output device is a serial device, send an UNLISTEN command on the serial bus.  
If the current input device is a serial device, send an UNTALK command on the serial bus.  
Set 99, the current input device, to be the keyboard.  
Set 9A, the current output device, to be the screen.

## **Reset to No Open Files F32F/F3EF-F332/F3F2**

**Called by:**  
Indirect JMP through (032C) from Kernal CLALL vector at FFE7.

Reset location 98, the number of open files, to zero and fall through to F333/F3F3 to reset any open serial channels and reset the default device numbers.

### **Operation:**

1. Set 98, the number of open files, to 0.
2. Fall through to F333/F3F3, Clear Serial Channels and Reset Default Devices routine.

## **CLOSE**

### **FFC3**

#### **Called by:**

JSR at E1CC/E1C9 in BASIC's CLOSE

#### **Entry requirements:**

Accumulator should contain the number of the logical file to be closed.

JMP(031C) with a default of F291/F34A.

If the logical file number in the accumulator is found in the logical file number table, also retrieve the current device number from the device number table and the secondary address from the secondary address table.

Then, execute the appropriate CLOSE routine for this current device.

If accessing a serial device, for secondary addresses < 128 (decimal), command the current device to LISTEN, send a CLOSE secondary address, and command the serial device to UNLISTEN. For secondary addresses > 128, this close sequence is omitted.

For an RS-232 device, bring the transmitted data line high, which is the idle state for RS-232 communications. Also, reset the pointers to the end of memory by reclaiming the space used for the RS-232 transmit and receive buffers.

When closing a logical tape file, determine whether writing to or reading from tape. If writing to tape then store a final byte of 0 in the tape buffer and write the buffer to tape.

For all types of devices, a common CLOSE exit is used. The number of open files, 98, is decremented, and the entry for this logical file is deleted from the logical file number table, the device number table, and the secondary address table.

### Determine Device for CLOSE F291/F34A-F2AA/F363

#### Called by:

Indirect JMP through (031C) from Kernal CLOSE vector at FFC3.

Upon entry, the accumulator contains the logical file number to be closed. First, it calls a routine to determine if the logical file number is in the logical file number table. If the number is not in the table, then it will exit with carry clear. If the number is in the table, then it will retrieve the current device number and secondary address corresponding to this file.

Push the current index into the tables corresponding to the logical file number onto the stack. Determine the type of device the logical file is using and branch to the appropriate routine for closing screen, keyboard, serial, or tape devices, or fall through to following code for closing RS-232 devices.

#### Entry requirements:

Accumulator should contain the number of the logical file to be closed.

#### Operation:

1. JSR F314/F3D4 to see if the logical file number is in the logical file number table. Return with X as the index into table corresponding to this logical file if it exists; return with Z = 1 (detected with BEQ) if the logical file is found.
2. If the file is not found CLC and RTS.
3. If the file is found, then JSR F31F/F3DF to retrieve the current device number, BA, the current secondary address, B9, and the current logical file number, B8, from the tables for these with entries corresponding to the location of current logical file number in the logical file number table.
4. Transfer index into tables to accumulator and push on stack for later retrieval by the individual CLOSE routines for RS-232 and serial devices.
5. If current device is the keyboard or screen, branch to F2F1/F3B1 to decrement number of open files and remove current file entry from the three tables.
6. If current device is a serial device, branch to F2EE/F3AE for a JMP to the routine to close a serial device.
7. If current device is tape, branch to F2C8/F38D to close tape files.

8. If current device is RS-232, fall through to the routine at F2AB/F364 to close an RS-232 device.

### **Common Exit for Close Logical File Routines F2F1/F3B1-F30E/F3CE**

#### **Called by:**

Falls through after JSR to Close Logical File for Serial Device at F2EE/FEAE, BEQ at F29F/F358 and F2A3/F35C in Determine Device to Close, BEQ at in F2CC/F391 Close Logical File for Tape, BNE at F2E4/F3A4 in Close Logical File for Tape, JMP at F2EB/F3AB in Close Logical File for Tape; alternate entry at F2F2/F3B2 by JSR at F2AC/F365 in Close Logical File for RS-232 Device.

The index into the file tables for the current logical file is retrieved from the stack (except for the alternate entry from RS-232 which has already pulled it from the stack).

The number of open files, 98, is decremented and compared to the index into the file tables. If equal, the current logical file is the last entry in the file table. In this case, there is no need to delete the actual entries in the tables since the pointer to the tables will now cause the next OPEN to overwrite these entries.

If the current logical file index is not equal to the number of open files (after the decrement), replace the current entries of the logical file number, device number, and secondary address tables with the last entries in the table. As the order within a particular table is unimportant, this rearrangement effectively deletes the current entries for the logical file, device, and secondary address.

#### **Entry conditions:**

Index into file tables for current logical file is pulled from the stack at entry.

#### **Operation:**

1. Pull index into file tables for current logical file from stack.
2. Transfer the index into the tables to X register.
3. Decrement the number of open files (plus one), location 98.
4. If X register equals the value in 98, the logical file being closed is the last entry in the tables. Since 98 points to the next available space in the tables, the next OPEN will overwrite the entries for this current logical file. Thus, just CLC and RTS.

5. If the number of open files (plus one) after decrement is not equal to the value in the X register, the current logical file being closed is not the last entry in the table. In this case, move the last entries in the three tables (device number, secondary address, logical file number) to the current logical file entries. Before this move, the last entry is pointed to by 98 and the current entry by the X register.
6. CLC and RTS.

### **CLRCHN FFCC**

#### **Called by:**

JSR at A447/C447 in BASIC's Error Message Handler, JSR at ABB7/CBB7 in BASIC's INPUT#, JSR at E37B/E467 in BASIC's Warm Start, JSR at F6F4/F777 in Test for STOP Key, JSR at F716/F799 in Error Message Handler.

JMP(0322) with a default of F333/F3F3.

If the current output device is a serial device, send an UNLISTEN command on the serial bus. If the current input device is a serial device, send an UNTALK command on the serial bus.

Set 99, the current input device, to be the keyboard.

Set 9A, the current output device, to be the screen.

### **Clear Serial Channels and Reset Default Devices F333/F3F3-F349/F409**

#### **Called by:**

Indirect JMP through (0322) from Kernal CLRCHN vector at FFCC, fall through from F331/F3F1 in Reset to No Open Files.

#### **Operation:**

1. If the current output device is a serial device, JSR EDFE/EF04 to command the serial device to unlisten.
2. If the current input device is a serial device, JSR EDEF/EEF6 to command the serial device to untalk.
3. Reset 9A, the current output device, to the screen (3).
4. Reset 99, the current input device, to the keyboard (0).

### GETIN FFE4

#### Called by:

JSR at E121 in BASIC's Get a Character.

#### Setup routines:

OPEN, CHKIN

JMP(032A) with a default of F13E/F1F5.

When retrieving characters from the keyboard, if any characters are in the keyboard buffer, the first character (an ASCII value) in the buffer is returned in the accumulator, and the rest of the characters are moved up one position in the buffer. If no characters are in the keyboard buffer, return with accumulator cleared to 0.

You would use GETIN to retrieve the first character in the keyboard buffer. Contrast this to CHRIN, which does not retrieve anything until RETURN is entered, then returns a character from the logical screen line.

If retrieving from device 2, RS-232, see if the RS-232 receive buffer contains any characters. If it is empty, return with accumulator set to 0. If it contains characters, return with accumulator containing next character in the receive buffer and increment the pointer into the receive buffer.

If retrieving from channel 3 (the screen), channels  $\geq 4$  (serial devices), or channel 1 (tape), do the same routines for GETIN that CHRIN does for these devices.

For screen GETIN, return the ASCII code for the screen character in the current logical line pointed to by D3, the column the cursor is on. D3 is then incremented to point to the next character in the line. If D3 is on the end of the line, return the ASCII code \$0D for return.

For serial GETIN, the accumulator returns the byte received over the serial bus. However, if any I/O status errors occur, return with accumulator containing \$0D.

For tape GETIN, return the next byte from the tape buffer. Also, read one byte ahead to see if the next byte is zero, indicating end of file, and if true, set end-of-file status in 90.

### **GETIN Preparation** **F13E/F1F5-F14D/F204**

#### **Called by:**

Indirect JMP through (032A) from Kernal GETIN vector at FFE4.

This routine first determines if the current input device is the keyboard. If not, GETIN falls through to F14E/F205 for an RS-232 device or branches to F166/F21D for other devices using the same routines as are used by CHRIN.

If the current input device is the keyboard and if the keyboard buffer contains characters, JMP E5B4/E5CF to retrieve the first character from the keyboard buffer.

#### **Operation:**

1. If 99, the current input device, is not 0 (the keyboard), branch to step 5.
2. If 99 is 0 for the keyboard, see if any characters are in the keyboard buffer as indicated by C6, the number of characters in the keyboard buffer.
3. If no characters are in the keyboard buffer, just CLC and RTS, thus returning with the accumulator set to 0.
4. If characters do exist in the keyboard buffer, disable IRQ interrupts and JMP E5B4/E5CF to retrieve the first character from the keyboard buffer and exit.
5. If the current input device, 99, is 2 for RS-232, fall through to F14E/F205 for the routine to get characters from RS-232.
6. If the current input device is neither 0 nor 2, branch to F166/F21D to get a character from other devices. F166/F21D is located in the Determine Input Device routine used by CHRIN; thus, other devices (tape, screen, serial) perform the same routines for both GETIN and CHRIN.

### **IOBASE** **FFF3**

#### **Called by:**

JSR at E09E/E09B BASIC's in RND  
JMP E500.

This routine returns (to the X and Y registers) the address of the start of the I/O registers that control the 6526 CIA/6522 VIA chips. You can write programs that refer to the I/O registers without knowing the exact address of the I/O



register. To write a program in this manner, you would call IOBASE to get the starting address of the I/O registers and add an index to the particular I/O register to which you are referring.

IOBASE for the VIC returns with X register set to \$10 and Y register set to \$91 (the address of the first register of VIA #1 is 9110). IOBASE for the 64 returns with X register set to \$00 and Y register set to \$DC (the address of the first register of CIA #1 is DC00). By calling IOBASE, you can determine both the starting address of the I/O registers and which computer the program is running on. Thus, your program could test which computer it is running on and read or write to the appropriate I/O register for this computer, giving you the ability to write one program that works on both the 64 and the VIC. You still will have to know what the CIA/VIA registers do and how to modify them, since what works for a VIA register does not necessarily work the same way on the corresponding CIA register.

BASIC's RND uses this routine to access the CIA/VIA timer registers in generating a random number.

### **IOINIT (64 only)**

**FF84**

#### **Called by:**

None.

JMP FDA3 to initialize the CIA registers, the 6510 I/O data registers, the SID chip registers, start CIA #1 timer A, enable CIA #1 timer A interrupts, and set the serial clock output line high.

The closest equivalent to this routine on the VIC is JMP FDF9 to initialize the 6522 registers, set VIA #2 timer 1 value, start timer 1, and enable timer 1 interrupts.

Since these routines are called during system reset, the main use for IOINIT is for an autostart cartridge that wants to use the same I/O settings that the Kernal normally uses.

### **LISTEN**

**FFB1**

#### **Called by:**

None.

JMP ED0C/EE17.

## Kernal Routines

---

At ED0C/EE17 the accumulator, which contains the device number, is ORed with \$20, turning on bit 5 to prepare to send a LISTEN command on the serial data output line. The device number should be 0–31 (decimal). (If you specify a value > 31, you mess up the high nybble which is used for sending commands on the serial bus.)

RS-232 interrupts are disabled.

Finally, the LISTEN command for this device is sent on the serial bus. To send the LISTEN, the 64/VIC brings the serial attention output line low to cause all devices on the serial bus to listen for a command coming on the bus.

### **LOAD FFD5**

#### **Called by:**

JSR at E175/E172 in BASIC's LOAD/VERIFY.

#### **Setup routines:**

SETLFS, SETNAM

#### **Entry requirements:**

Accumulator should be set to 0 for LOAD; accumulator set to 1 for VERIFY.

If relocatable load desired: Set X register to the low byte of load starting address, and Y register to the high byte of load starting address.

JMP F49E/F542 to store the X register and Y register in (C3), the starting address of the load, and then JMP(0330) with a default of F4A5/F549.

At F4A5/F549, determine the device. The keyboard, screen, and RS-232 are illegal devices.

For a serial device you must specify a filename. If you don't, the MISSING FILE NAME error message is displayed. With a valid filename, the computer commands the current serial device to listen and sends the secondary address of \$60, indicating a load, followed by the filename. Then it tells the device to unlisten. Next, it tells the current serial device to talk, sends the current secondary address of \$60, and receives a byte from the serial bus. If the I/O status word indicates the

byte was not returned fast enough, a read time-out has occurred and the FILE NOT FOUND error message is displayed. The first two bytes received from the serial device are used as a pointer to the start of the load area (AE). However, if a secondary address of 0 is specified at entry to load, the X and Y registers stored in (C3) at entry are used as the starting address of the load—thus providing for a relocatable load. Then it receives bytes from the serial bus and stores or verifies them until the EOI status is received. Once the EOI status is received, the serial device is commanded to untalk, and the serial device sends the last buffered character. The serial device then is sent a CLOSE and told to untalk.

For tape LOAD/VERIFY, the LOAD routine first checks if the tape buffer is located in memory  $\geq 0200$ . If so, it loads the tape buffer with a header retrieved from the tape. If a filename has been specified, a specific header with this filename is loaded; if there is no filename, it loads the next header on the tape. Only tape headers with tape identifiers of 1 or 3 are acceptable for LOAD/VERIFY. A tape identifier of 5 indicates an end-of-tape header, and in this case the routine will exit with carry set and accumulator set to 5. Tape identifiers of 2 or 4 are for sequential files.

A tape identifier of 3 causes a nonrelocatable load even if you have specified values in the X and Y registers at entry and a secondary address of 0. That is, you can't override a tape identifier of 3—it forces a nonrelocatable load.

A tape identifier of 1 allows a relocatable load. If the tape identifier is 1 and the secondary address is 0, the X and Y register values at entry are used to determine the starting address for the load.

For a nonrelocatable load, the starting address for the load is taken from the tape header. The ending address for the load (in both relocatable and nonrelocatable loads) is determined by adding the length of the program to the starting address. After determining whether to do a relocatable or nonrelocatable load, it loads RAM from the next two program blocks on tape (two blocks are used for error correcting purposes; they should be identical copies of each other).

### **Jump to LOAD Vector** **F49E/F542-F4A4/F548**

#### **Called by:**

JMP from Kernal LOAD vector at FFD5.

The starting address for a possible relocatable load, specified in the X and Y registers at entry, is stored in (C3), followed by JMP (0330) with the default vector of F4A5/F549.

#### **Operation:**

1. STX C3.
2. STY C4.
3. JMP(0330).

### **Determine Device for LOAD** **F4A5/F549-F4B7/F55B and F533/F5CA-F538/F5D0**

#### **Called by:**

Indirect JMP through (0330) at F4A2/F546 in Jump to LOAD Vector.

This routine determines which device is being used for the load. Invalid devices are the screen, keyboard, or RS-232. Valid devices are serial devices or tape, and for these the routine passes control to the appropriate serial or tape load routine.

#### **Operation:**

1. STA 93, thus setting the LOAD/VERIFY flag to 0 for LOAD or to 1 for VERIFY.
2. Reset 90, the I/O status, to 0.
3. LDA from BA, the current device number.
4. If the current device is the keyboard (0) or the screen (3), JMP F713/F796 to display the ILLEGAL DEVICE NUMBER error message.
5. If the current device number is less than 3, branch to step 7.
6. If the current device number is greater than 3, it's a serial device. Fall through to F4B8/F55C to load from a serial device.
7. If the current device is RS-232, JMP F713/F0B9 to display ILLEGAL DEVICE NUMBER.
8. If the current device number is not 2 (RS-232), the only number left is 1 (tape). Fall through to F539/F5D1 to load from tape.

## **MEMBOT FF9C**

### **Called by:**

JSR at E403/E3E5 in BASIC's Cold Start.

### **Entry requirements:**

Carry should be set or clear, depending on function desired:

Set carry to read bottom of memory.

Clear carry to set bottom of memory. The X register is the low byte of the address of the bottom of memory, and the Y register is the high byte of the address of the bottom of memory.

JMP FE34/FE82.

If the carry is clear at entry, set the pointer to bottom of memory (0281) from X and Y registers.

If the carry is set at entry, load X and Y registers from (0281), the pointer to the bottom of memory.

The initial values of (0281) are 1000 for an unexpanded VIC, 0400 for a VIC with 3K expansion, 1200 for a VIC with 8K or more expanded, and 0800 for the 64.

## **MEMBOT Execution FE34/FE82-FE42/FE90**

### **Called by:**

JMP from Kernal MEMBOT vector at FF9C.

If the carry is clear at entry, set (0281), the pointer to the bottom of memory, from the X and Y registers. If carry is set at entry, load X and Y registers from (0281).

### **Operation:**

1. If carry is clear, branch to step 3.
2. Load X and Y registers from pointer to bottom of memory (0281), and fall through to step 3.
3. Set (0281) from values in X and Y registers.
4. RTS.

## **MEMTOP FF99**

### **Called by:**

JSR at E40B/E3ED in BASIC's Cold Start.

### Entry requirements:

Carry should be set or clear, depending on function desired:

Set carry to read end of memory.

Clear carry to set end of memory. The X register contains the low byte of the address of the start of memory, and the Y register contains the high byte of the address of the start of memory.

JMP FE25/FE73.

If carry is clear at entry, set pointer to top of memory (0283) from X and Y registers.

If carry is set at entry, load X and Y registers from (0283), the pointer to the top of memory.

### MEMTOP Execution

**FE25/FE73-FE33/FE81**

#### Called by:

JMP from Kernal MEMTOP vector at FF99; alternate entry at FE27/FE75 by JSR at F2B2/F377 in Close Logical File for RS-232, JSR at F468/F527 in Open RS-232 Device; alternate entry at FE2D/FE7B by JMP at F480/F53F in Open RS-232 Device, JMP at FDCF in Initialize Memory Pointers (VIC only).

If entering at FE25/FE73, the carry flag determines whether the top of memory is being set or read. If the carry bit is clear, or if the routine is entered at FE2D/FE7B, the top of memory pointer (0283) is set from the X and Y register values. If the carry is set, or if the routine is entered at FE27/FE75, the X and Y registers are set from the top of memory pointer (0283).

#### Operation:

1. FE25/FE73: If carry is set, branch to step 3.
2. FE27/FE75: Load X and Y registers from pointer to top of memory (0283), and fall through to step 3.
3. FE2D/FE7B: Set (0283) from values in X and Y registers.
4. RTS.

### OPEN

**FFC0**

#### Called by:

JSR at E1C1/E1BE in BASIC's OPEN.

### Setup routines:

SETLFS, SETNAM

JMP(031A) with a default of F34A/F40A.

OPEN checks whether another logical file can be opened. Another logical file can be opened if the logical file number is not 0 and if fewer than ten logical files are already open. OPEN exits if trying to open to the screen or keyboard, as these devices do not use files.

For a serial device, OPEN commands the serial device to listen and then sends a secondary address for OPEN to this serial device.

For tape, OPEN checks for a tape header of a sequential file if reading, or writes a tape header for a sequential file if writing.

The RS-232 OPEN initializes various RS-232 lines and creates two 256-byte buffers at the top of memory. RS-232 OPEN handles the x-line handshaking opening sequence incorrectly on the VIC.

### OPEN Execution

**F34A/F40A-F3D4/F494**

#### Called by:

Indirect JMP through (031A) from Kernal OPEN vector at FFC0.

OPEN creates a logical file that can be used by Kernal I/O routines.

OPEN first checks if the logical file number specified is 0. If it is 0, jump to the display the NOT INPUT FILE message and exit, as a logical file number of 0 is not permitted.

If the number of files already open is less than ten and the logical file specified is not yet open, create entries in the logical file number table, device number table, and secondary address tables for this file. If a file already exists that uses the specified file number, the FILE OPEN error message is displayed. If there are already ten open files, the TOO MANY FILES message is displayed.

If the device is the screen or keyboard, exit as these devices do not use files. For RS-232-C, serial, or tape devices, jump or branch to their respective OPEN routines.

### Operation:

1. If the logical file number is 0, JMP F70A/F78D to set error message number of 6 (NOT INPUT FILE) and exit.
2. JSR F30F/F3CF to see if the logical file number in B8, the current logical file, is already present in the logical file number table.
3. If the logical file number is in the logical file number table, JMP F6FE/F781 to set the error message number of 2 (FILE OPEN) and exit.
4. LDX 98 to get the number of open files.
5. If less than ten open files exist, continue with step 6. If ten files are already open, JMP F6FB/F77E to set error message number of 1 (TOO MANY FILES).
6. Increment the number of open files, 98.
7. Store the current logical file number in the logical file number table, using the X register value from step 4 as an index into the table at 0259.
8. LDA B9, the secondary address, then ORA \$60 in case a secondary address is to be sent to a serial device. Store this value in the secondary address table entry that corresponds to the entry for this file in the logical file number table.
9. Store the current device number, BA, in the device number table entry that corresponds to the entry for this file in the logical file number table.
10. If the current device is the keyboard or the screen, CLC and RTS as there is no need to open files to these devices.
11. If the current device is 1 or 2, branch to step 14. For devices > 3, fall through to step 12.
12. JSR to F3D5/F495 to send secondary address in B9 to the current device on the serial bus. The secondary address is ORed with \$F0 before sending to provide the secondary address for OPEN. Return with carry clear if the serial device was present and the secondary address was sent without errors. No error routine is called from OPEN if this sequence fails.
13. If the carry is clear on return from the JSR in step 12, exit with carry clear. However, if the routine does not return the carry clear to indicate the device was present and responded in time, OPEN does not branch to any error routine. This appears to be why you can open nonexistent



devices without getting an error until you actually try to send or receive data for the device.

14. If the device number is 2 (RS-232), JMP F409/F4C7 to open an RS-232 device and RTS upon completion of the RS-232 OPEN.
15. If the device number is none of the above, it must be 1 (tape). Branch to F38B/F44B to open a logical file to tape.

### **PLOT**

**FFF0**

#### **Called by:**

JSR at AAE9/CAE9 in BASIC's Tab to Column for PRINT, JSR at AAFA/CAFA in BASIC'S TAB and SPC, JSR at B39F/D39F in BASIC'S POS.

#### **Entry requirements:**

Carry bit should be set or clear, depending on function desired:

Set carry to read cursor location (X register = row, and Y register = column).

Clear carry to set cursor location (X register = row, and Y register = column).

JMP E50A (see screen routines in chapter 7).

If the carry bit is clear at entry, move the cursor to the specified location. The contents of the the X register determine the new cursor row and the contents of the Y register determine the new cursor column.

If the carry bit is set at entry, read the cursor location and place the row value for the current cursor location into the X register and column value for the current cursor location into the Y register.

The row number indicates the physical line, while the column number indicates the column within a logical line. Valid physical line numbers in decimal are 0-24 (64) and 0-22 (VIC). Valid logical column numbers in decimal are 0-79 (64) and 0-87 (VIC).

### **RAMTAS (64 only)**

**FF87**

#### **Called by:**

None.

JMP FD50 to the Initialize Memory Pointers routine on the 64.

At FD50 the routine stores \$00 in locations 02-0101 and 0200-03FF; sets the pointer to the tape buffer, (B2), to 033C; sets the pointer to the end of RAM + 1 in (0283), sets the pointer to the start of RAM in (0281), sets the screen memory page to \$04.

Although the VIC does not have a RAMTAS Kernal vector, the corresponding operation on the VIC is done by JMP FD8D. At FD8D the routine stores \$00 in 00-FF and 0200-03FF; sets pointer to tape buffer, (B2), to 033C; sets the pointer to the end of RAM + 1 in (0283); sets the pointer to the start of RAM in (0281); sets the screen memory page to \$1E or \$10 depending on where RAM ends.

The RAMTAS routine would mainly be used by an auto-start cartridge since the RAMTAS functions are normally executed during system reset.

### **RDTIM FFDE**

#### **Called by:**

JSR at AF84/CF84 in BASIC's TI and TI\$.

JMP F6DD/F760.

This routine reads the jiffy clock (A2-A0) into the accumulator, X register, and Y register.

A0 is updated every 1/60 second. When the jiffy clock reaches a value equal to 24 hours, it is reset to 0.

#### **Exit conditions:**

Accumulator holds high byte of jiffy clock. X register holds middle byte of jiffy clock. Y register holds low byte of jiffy clock.

### **RDTIM/SETTIM Execution F6DD/F760-F6EC/F76F**

#### **Called by:**

JMP from Kernal RDTIM vector at FFDE; alternate entry at F6E4/F767 by JMP from Kernal SETTIM vector at FFDB.

From the RDTIM entry point, this routine reads the jiffy clock at A2-A0 into the accumulator, X register, and Y register, and then falls through to the following routine at F6E4/F767.

## Kernal Routines

---

If entering at the SETTIM entry point at F6E4/F767, set the jiffy clock at A2, A1, and A0 from the accumulator, X register, and Y register.

### Operation:

1. F6DD/F760: SEI to disable interrupts.
2. LDA from A2, LDX from A1, and LDY from A0.
3. F6E4/F767: SEI to disable interrupts (which has no effect if interrupts were already disabled in step 1).
4. STA at A2, STX at A1, and STY at A0.
5. CLI to enable interrupts, then RTS.

## READST

### FFB7

#### Called by:

JSR at ABDD/CBDD in BASIC's INPUT, JSR at AF9A/CF9A in BASIC's STATUS, JSR at E180/E17D E195 in BASIC's LOAD/VERIFY.

JMP FE07/FE57 to read the I/O status word, 90, returning the value in the accumulator. This value reflects certain conditions during serial or tape I/O.

The 64/VIC *Programmer's Reference Guides* contain some errors:

First, when VERIFYing for a serial device, a VERIFY error can occur.

Second, for the VIC you cannot read the RS-232 status register, 0297, by calling this routine. READST for RS-232 always returns zero on the VIC. If you want to read the RS-232 status on the VIC, read 0297 directly; don't call this routine. This error in READST is corrected in the 64.

Third, detecting an end-of-tape header allows BASIC to display the DEVICE NOT PRESENT error message, but the Kernal routines for OPEN or LOAD/VERIFY do not set location 90. Thus, READST will not return the end-of-tape status condition following OPEN or LOAD/VERIFY. You can check end-of-tape status upon return from OPEN or LOAD/VERIFY by checking for the carry bit set and the accumulator set to 5, which are the conditions that indicate end-of-tape.

The table below shows the possible values returned by READST:

## Kernal Routines

### READST Values

Hex Value	Bit	Serial I/O	Tape Read/ LOAD/VERIFY	RS-232 (64 only)
\$80	7	Device not present		Break detected
\$40	6	EOI status	End of file	DSR signal missing
\$20	5		Checksum error	
\$10	4	VERIFY error	Unrecoverable read error	CTS signal missing
\$08	3		Long block	Receiver buffer empty
\$04	2		Short block	Receiver buffer overrun
\$02	1	Read timeout		Framing error
\$01	0	Write timeout		Parity error

### Status Terms

**Long Block:** Tape read is trying to read data bytes after the first block has already completed.

**Short Block:** Tape read is reading leader bits between blocks while the byte action routine is still expecting to be reading bytes from the block.

**Unrecoverable Read Error:** During tape read and LOAD/VERIFY, more than 31 errors were detected in block 1. This is also set if read or VERIFY errors for the same byte occurred in both blocks 1 and 2.

**Checksum Error:** Computed parity for the loaded area is not the same as the final byte of tape block 2 (the parity computed during the SAVE of the second block).

**End of File:** This status is set when doing CHRIN from tape for a sequential file and the read-ahead byte in the tape buffer is 0.

**VERIFY Error:** The byte retrieved from the serial device does not match the byte in memory.

**EOI (End or Identify):** This is set during the Receive Byte from Serial Device routine when the EOI handshake is performed. Set during serial read to indicate the last byte has been sent from the serial device. The unusual term EOI is a holdover from the IEEE-488 bus definitions used on older PET/CBM computers; you may find it simpler just to remember this as End of File for disk.

**Device Not Present:** Device does not respond with the proper handshake sequence during OPEN, LOAD, VERIFY, or SAVE operations.

**Read Timeout, Write Timeout:** Read or write timeouts are set when the serial device doesn't handshake within the allocated time.

**Break Detected:** This is set if the check for a stop bit finds a 0 rather than a 1, and the data bits received so far are all 0's.

**Framing Error:** This is set if the check for a stop bit finds a 0 and the data bits received so far included some bits set to 1.

**DSR Signal Missing:** The 64 can't detect the Data Set Ready signal from the RS-232 device during x-line handshaking.

**CTS Signal Missing:** The 64 can't detect the Clear To Send signal from the RS-232 device during x-line handshaking.

**Parity Error:** The parity bit indicates an error in transmission of this byte.

**Receiver Buffer Empty:** Nothing is in the RS-232 input buffer. This allows routines to test the status so they don't loop waiting for data.

**Receiver Buffer Overrun:** The RS-232 input buffer is full and another byte has been received.

### Read/Set Status and Set Message Control FE07/FE57-FE20/FE6E

#### Called by:

JMP from Kernal READST vector at FFB7; alternate entry at FE18/FE66 by JMP from Kernal SETMSG vector at FF90; alternate entry at FE1C/FE6A by JSR at EDB2/EEB9 in Set Status Word, JSR at EE4F/EF4D in Receive Byte from Serial Device, JSR at F18A/F241 in CHRIN from Tape, JSR at F518/F5AF in Load/Verify from Serial Device, JSR at FA81/FACE in Determine Action for Byte, JSR at FAC6/FB13 in Test for Short Block, JSR at FB35/FB82 in Flag Unrecoverable Read Error, JSR at FB8B/FBCC in Reset Vectors and Compute Checksum.

If entering at FE07/FE57, determine the device number. For an RS-232 device, the VIC always stores 0 in the RS-232 status word, 0297, and returns with accumulator set to 0, while the 64 correctly returns with the accumulator holding the value from 0297, and also stores 0 in 0297. For other devices, return with the value from 90, the I/O status, in the accumulator.

If entering at FE18/FE66, store the contents of the accumulator in 9D, the Kernal message control flag. \$80 sets Kernal control messages on, \$40 sets Kernal error messages on, \$C0 sets both Kernal control and error messages on, and

## Kernal Routines

\$00 turns off all Kernal messages. Exit with the accumulator containing the value from 90, the I/O status word.

If entry at FE1A/FE6A, then set 90, the I/O status, by ORing the accumulator with the current value of 90. The routines that call this entry point set the accumulator to a value for an I/O status.

### Operation:

1. FE07/FE57: If the device number, BA, is not 2, branch to step 4.
2. If device number is 2 (RS-232): LDA 0297, PHA (64 only), LDA \$00, STA 0297, PLA (64 only), RTS.
3. FE18/FE66: Store accumulator in 9D, the Kernal message control flag.
4. LDA 90, the I/O status.
5. FE1C/FE6A: ORA 90. Thus, if the routine continues from the instructions above, 90 is not changed. However, if the alternate entry point of FE1C/FE6A is used, a new value will result from this ORA.
6. STA 90, updating the I/O status word if the alternate entry point of FE1C/FE6A was used.

### RESTOR FF8A

#### Called by:

None.

JMP FD15/FD52 to execute the routine to initialize the Kernal RAM vectors. This RAM vector initialization is also done during system reset.

Calling this routine restores the vectors at (0314)–(0332) to their default values from the table at FD30/FD6D.

### SAVE FFD8

#### Called by:

JSR at E15F/E15C in BASIC's SAVE.

#### Setup routines:

SETLFS, SETNAM (not required for saving to tape)

#### Entry requirements:

The accumulator should contain the offset within zero page to

## Kernal Routines

---

a two-byte pointer to the start of the area to be saved. The X register should hold the low byte of the address of the end of the area to be saved + 1. Y register should hold the high byte of the address of the end of the area to be saved + 1.

JMP F5DD/F675 to save memory to a serial device or to tape. Saves to the screen, keyboard, or RS-232 are not permitted.

If saving to tape from the VIC, only the contents of memory locations 0-7FFF may be saved. This restriction does not apply when saving to tape from the 64.

A filename is required (through SETNAM) when saving to serial devices; a filename is optional when saving to tape.

At F5DD/F675, the routine loads the pointer to the end of the save area + 1, (AE), from the X and Y registers. (End + 1 denotes the fact that you must load X and Y to point to the location just past the end of the save area, since the save routines consider the save complete when the pointer to the save area equals the value of the pointer (AE).) It also sets (C1), the pointer to the start of the save area, from the zero page pointer indexed by the accumulator, and then performs an indirect JMP through the vector at (0332), which defaults to F5ED/F675.

For a serial save, the routine commands the current serial device to listen with attention, then sends a secondary address of \$61 to indicate a SAVE operation. If the device is present, the filename and the starting address are sent to the serial device. Next, the routine sends all the bytes from the save area over the serial bus. When the save is complete, it sends a secondary address of \$E1 to indicate the CLOSE command and commands the serial device to unlisten.

For tape save, it is important that you specify the secondary address correctly. For an even secondary address, the header for the saved program will have a identifier byte of 1, indicating a relocatable program. An odd secondary address produces a header identifier byte of 3, indicating a non-relocatable program. Also, if you have bit 1 on in the secondary address (\$02 or \$03 would set bit 1), then an end-of-tape header with a identifier byte of 5 is written following the saved program.

The tape save operation first writes a header to tape. This tape header contains the identifier byte, the starting address and ending address + 1 of the save area, and the filename (if a filename is used). Then data from the save area is written to

tape. If bit 1 of the secondary address is 1, an end-of-tape header is also written following the data from the save area. Two identical copies of the tape header(s) and the program are written to tape to allow for error checking and correction during tape loading.

### **Jump to SAVE Vector** **F5DD/F675-F5EC/F684**

#### **Called by:**

JMP from Kernal SAVE vector at FFD8.

Set the pointer to the end of the save area + 1, (AE), from the X and Y registers.

The accumulator value at entry is transferred to the X register and is used as an index into page zero for the location of two bytes that specify the starting address for the save. Set the pointer to the start address of the save area, (C1), from these two page-zero bytes.

Jump to the address in the vector at (0322), normally F5ED/F685.

#### **Operation:**

1. STX AE, the low byte of the address of the end of the save area + 1.
2. STY AF, the high byte of the address of the end of the save area + 1.
3. TAX and LDA 00,X to get the low byte of the address of the start of the save area, and STA in C1.
4. LDA 01,X to get the high byte of the address of the start of the save area, and STA in C2.
5. JMP (0322) to the save routine. The default address in the vector is F5ED/F685.

### **Determine Device for SAVE** **F5ED/F685-F5F9/F691**

#### **Called by:**

Indirect JMP through (0322) at F5EA/F682 in Jump to SAVE Vector.

If the current device is the keyboard or the screen, load the accumulator with 9 and set the carry bit to display the IL-LEGAL DEVICE NUMBER message, then exit.



If the current device is either tape (1) or RS-232 (2), branch to Control Routine for Tape Save (which treats RS-232 as an illegal device).

If the current device is a serial device, fall through to the Save to Serial Device routine.

### **Operation:**

1. If the current device is the keyboard or the screen, JMP F713/F796 to display the ILLEGAL DEVICE NUMBER error message, set accumulator to 9, set carry, and exit. The keyboard or the screen is not a valid device for saves.
2. If the current device is RS-232 or tape, branch to F659/F6F1, a routine that determines whether the save is to RS-232 or tape. If RS-232 is specified, the ILLEGAL DEVICE NUMBER message is displayed. If the device is a tape device, the tape save routines are executed.
3. If the device is none of the above, its device number must be  $\geq 4$ ; thus, it's a serial device. Fall through to Save to Serial Device routine at F5FA/F692.

## **SCNKEY**

**FF9F**

### **Called by:**

None.

JMP EA87/EB1E to the Keyboard Scan routine (see chapter 4) to check for a keypress. If a valid key is found down and the keyboard buffer is not full, the ASCII code value for the key is placed in the buffer.

SCNKEY is useful if you have written a machine language program that runs with IRQ interrupts disabled, but you still want to scan the keyboard.

## **SCREEN**

**FFED**

### **Called by:**

None.

JMP E505 to return the number of columns on the standard display screen in the X register and the number of rows in the Y register. On the 64, the routine returns 40 in X and 25 in Y. The VIC routine returns 22 in X and 23 in Y.

A definitive way to let a program know whether it's running on the VIC or the 64 is to JSR to SCREEN and test the values returned.

### **SECOND FF93**

**Called by:**  
None.

**Setup routines:**  
LISTEN

JMP EDB9/EEC0 to send the byte in the accumulator on the serial bus as a secondary address command with the serial attention output line set low. After this command is sent, the serial attention output line is brought high, the setting for transmitting normal data bytes.

You must ORA the secondary address with \$60 before calling this routine to convert the secondary address to a recognized IEEE secondary address command. See page 378 of Raeto Collin West's *Programming the PET/CBM* for a detailed chart of the IEEE command groups.

### **SETLFS FFBA**

**Called by:**  
JSRs at E1DD/E1DA, E1F0/E1ED, and E1FD/E1FA in BASIC's Set LOAD/VERIFY/SAVE Parameters; JSRs at E22B/E228, E23F/E23C, and E24E/E24B in BASIC's Handle Parameters for OPEN and CLOSE.

#### **Entry requirements:**

The accumulator should hold the logical file number, the X register should hold the device number, and the Y register should hold the secondary address.

JMP FE00/FE50 to set the logical file number, device number, and secondary address for a subsequent open, load, or save.

The logical file number can be 1-255.

The device numbers can be 0-31. Assigned device numbers include 0 for the keyboard, 1 for tape, 2 for RS-232, 3 for the screen, and 4-31 for serial bus devices. By convention, se-

rial device numbers 4 and 5 are usually used for printers and 8–11 for disk drives.

See the comments in the paragraphs on SAVE and LOAD routines about secondary addresses. An even secondary address gives a identifier byte of 1 for a relocatable program tape header. An odd secondary address gives a tape identifier of 3 for a nonrelocatable program tape header. A secondary address that has bit 1 on (e.g., \$02 or \$03) produces an end-of-tape header with an identifier byte of 5.

Secondary addresses  $\geq 128$  (decimal) will not be sent on the serial bus. For reading from serial, use an even secondary address. For writing to serial, use an odd secondary address. Valid secondary addresses for serial devices are 0–31 (decimal). If you specify a higher value, you may be sending a command other than what you intended, since secondary addresses greater than 31 are used to represent commands to serial devices.

### **Set Logical File Number, Device Number, Secondary Address**

**FE00/FE50–FE06/FE56**

#### **Called by:**

JMP from Kernal SETLFS vector at FFBA.

In preparation for other Kernal I/O routines, this sets the logical file number, device number, and secondary address.

Device numbers should range only from 0–31 (decimal). Serial commands such as OPEN, TALK, LISTEN, CLOSE, and others use bits 5–7 of the secondary address to specify the type of command. The commands do this ORA of the command high nybble with the device number. Thus, a device number greater than 31 might result in an invalid command after the ORA.

Also, if you specify a device number  $> 31$ , you do not get a device error until you actually try to send the device data (or retrieve data from it). This quirk is due to OPEN not checking for an invalid carry status after opening for a serial device.

Normally, a secondary address  $\geq 128$  (decimal) indicates no secondary address is desired. However, for tape operations, 255 (or any odd value  $> 128$ ) is the same as a secondary address of 3. See “Block SAVE and LOAD” by Sheila Thornton in *COMPUTE!’s Second Book of VIC* from COMPUTE! Books.

### Operation:

1. STA B8, the logical file number.
2. STX BA, the device number.
3. STY B9, the secondary address.

### SETMSG

FF90

### Called by:

JSR at A47D/C47D in BASIC's Enable Kernal Control Messages, JSR at A874/C874 in BASIC'S Disable Kernal Control Messages.

### Entry requirements:

Accumulator should contain the value used to set message control: \$80 allows Kernal control messages; \$40 allows Kernal error messages; \$C0 allows both Kernal control and error messages; \$00 disallows all Kernal messages.

JMP FE18/FE66. This routine is called to determine which messages will be displayed in response to control or error conditions. The accumulator value at entry determines the setting of the message control status.

Thanks to Russ Davies for pointing out that bits 6 and 7 are reversed in describing how to set message control in the 64 and VIC *Programmer's Reference Guides*.

### SETNAM

FFBD

### Called by:

JSR at E1D6/E1D3 in BASIC's Set LOAD/VERIFY/SAVE Parameters, JSRs at E21B/E218 and E261/E25E in BASIC's Handle Parameters for OPEN and CLOSE.

### Entry requirements:

Accumulator should contain the length of the filename. The X register should hold the low byte of the starting address of the filename. The Y register should hold the high byte of the filename address. The filename may be stored at any addressable memory location.

JMP FDF9/FE49 to prepare a filename for subsequent OPEN, LOAD/VERIFY, or SAVE processing. The accumulator value, the length of the filename, is stored in B7. The pointer to the filename from the X and Y registers is stored in (BB).

Although you could create a filename that is 255 (decimal) characters long (the accumulator can hold a maximum value of \$FF or decimal 255), not all of this maximum filename size can be used.

For tape, the filename is stored in the tape buffer, which is 192 bytes long. However, 5 bytes are taken for the identifier and the starting and ending addresses, which leaves 187 bytes that can be used for the filename.

One quirk with the serial devices is that if the secondary address you specify in SETLFS is larger than 128, the filename is not sent for OPEN, LOAD, or SAVE.

### **Set Filename Location and Number of Characters FDF9/FE49-FDFF/FE4F**

#### **Called by:**

JMP from Kernal SETNAM vector at FFBD.

The location of the filename is placed in a pointer at (BB), and the number of characters in the filename is placed in B7.

This routine sets filename information for later use of the Kernal routines OPEN, SAVE, and LOAD. If no filename is needed for these routines, load the accumulator with zero before calling this routine. However, loading or saving to a serial device requires that a filename be present.

#### **Operation:**

1. STA B7, the number of characters in the filename.
2. STX BB, the low byte of the address of the filename.
3. STY BC, the high byte of the address of the filename.

### **SETTIM FFDB**

#### **Called by:**

JMP at AA1A/CA1A in BASIC's TI\$.

#### **Entry requirements:**

The accumulator should hold the high byte to be stored in the jiffy clock. The X register should hold the middle byte to be stored in the jiffy clock. The Y register should hold the low byte to be stored in the jiffy clock.

JMP F6E4/F767 to set the three-byte jiffy clock at A2-A0 from the values in the accumulator, X register, and Y register.

## SETTMO FFA2

### Called by:

None.

JMP FE21/FE6F to store accumulator in 0285. The VIC-20 *Programmer's Reference Guide* refers to this routine as setting a serial timeout flag and the Commodore 64 *Programmer's Reference Guide* refers to it as setting a flag for IEEE timeout. However, neither BASIC nor the Kernal refers to this vector. Since 0285 is not a register for an I/O chip and it is never referred to, it's hard to see how it can be used to enable or disable timeouts.

## STOP FFE1

### Called by:

JSR at A82C/C82C in BASIC's Test for STOP Key, JSR at F4F9/F590 in Load/Verify from Serial Device, JSR at F62E/F6C6 in Save to Serial Device; JSR at F8D0/F94B Test for STOP Key During Tape I/O; JSR at FE61/FECD in NMI Interrupt Handler (to find STOP and RESTORE).

JMP (0328) with a default of F6ED/F770. At F6ED/F770, test 91 for the value \$7F/\$FE. Location 91 contains the key switch value of the STOP key column (column seven/three) of the keyboard scan. If \$7E/\$FE is found, set the Z flag of the status register to 1, call FFCC to reset I/O channels, and set C6, the number of characters in the keyboard buffer, to 0.

If \$7E/\$FE is not found, the Z flag will be 0 on exit (BNE condition). In this case, the accumulator can still be tested for the keys shown below using the value shown following it.

### STOP Routine Return Values

Commodore 64 Key	Accumulator	VIC-20 Key	Accumulator
1	\$FE	Cursor down	\$7F
Left arrow	\$FD	/	\$BF
CTRL	\$FB	,	\$DF
2	\$F7	N	\$EF
Space	\$EF	V	\$F7
Commodore	\$DF	X	\$FB
Q	\$BF	Left SHIFT	\$FD

If no key is down in the STOP column, the routine returns \$FF in the accumulator (64 and VIC).

## **Test for STOP Key** **F6ED/F770-F6FA/F77D**

### **Called by:**

Indirect JMP through (0328) from Kernal STOP vector at FFE1.

This routine is called to test whether the STOP key is being held down. When the STOP key is found down, this routine exits with the Z status flag set to 1, allowing the calling routine to test for this result with BEQ.

Location 91 has the value of the keyboard scan for the STOP key column during the last IRQ or NMI interrupt.

### **Operation:**

1. LDA 91.
2. Check for the value that indicates the STOP key is pressed, \$7F/\$FE.
3. If STOP key is not pressed, then branch (BNE) to step 7 to RTS with the accumulator containing last value in \$91.
4. If STOP key is pressed, then JSR FFCC (the Kernal CLRCHN vector) to clear serial I/O and reset default input and output devices, returning with accumulator cleared to 0.
5. STA C6, the number of characters in the keyboard buffer, thus clearing the buffer.
6. Restore the status of the comparison from step 2, thus restoring Z to 1 (BEQ condition).
7. RTS.

## **TALK** **FFB4**

### **Called by:**

None.

### **Entry requirements:**

Accumulator should hold the serial device number (4-31 decimal).

JMP ED09/EE14 where the accumulator is ORed with \$40 to set the value for a talk command to the device. Send this command over the serial data bus while the serial attention output line is held low.

## TKSA

FF96

### Called by:

None.

### Setup routines:

TALK.

### Entry requirements:

Accumulator should hold the secondary address 0–31 (decimal) ORed with \$60.

JMP EDC7/EECE to send the secondary address after the TALK command and to do the TALK-LISTEN turnaround where the serial device becomes the talker and the 64/VIC (and other devices on the serial bus) become the listener.

The IEEE convention is that \$6X and \$7X represent secondary addresses. The 0–31(decimal) that you ORA with \$60 before calling TKSA results in \$6X if the accumulator holds 0–15 (decimal) and \$7X if the accumulator holds 16–31 (decimal). The VIC-1541 *User's Manual* states that the secondary addresses can be 2–15 with 15 the command channel and 0 and 1 reserved for load and save.

## UDTIM

FFEA

### Called by:

JSR at EA31/EABF in IRQ Interrupt Handler.

JMP F69B/F734 to update the jiffy clock at A2–A0 and store a value from the keyboard row for column number seven/three (which contains the STOP key) in 91 if a key in that row is detected.

Normally, this routine is called by the IRQ interrupt handler (64 and VIC) or by the NMI interrupt handler (VIC only). However, if you run a program with IRQ interrupts disabled, you should call this routine if you want the jiffy clock incremented and the STOP key column value saved in 91.

## UNLSN

FFAE

### Called by:

None.



JMP EDFF/EF04 to send \$3F, the command for UNLISTEN, over the serial bus. Serial devices that are listening should recognize the command and terminate their connection to the serial bus.

### **UNTALK FFAB**

**Called by:**

None.

JMP EDEF/EEF6 to send \$5F, the command for UNTALK, over the serial bus. Serial devices that are talking should quit talking and terminate their connection to the serial bus.

### **VECTOR FF8D**

**Called by:**

None.

**Entry requirements:**

Carry should be set or clear, depending on the function desired:

Set the carry bit to store the RAM vectors at (0314)–(0332) at the location pointed to by the X and Y registers.

Clear the carry bit to load the RAM vectors at (0314)–(0332) from the location pointed to by X and Y.

JMP FD1A/FD57 (see chapter 2).

Store X and Y at (C3), the base address of where the vector table will be read from or stored to.

If the carry is set, store the RAM vectors at (0314)–(0332) to the location pointed to by the X and Y registers.

If the carry is clear, load the RAM vectors at (0314)–(0332) from the location pointed to by X and Y.



## **Chapter 6**

---

# **Miscellaneous Routines**



# Miscellaneous Routines

---

## Set I/O Defaults and Home Cursor E59A/E5B5-E59F/E5BA

### Called by:

None.

This routine is not called by any other Kernal or BASIC routines. However, if you call it from a program, it calls a routine to reset the default input device to be the keyboard, the default output device to be the screen, homes the cursor, and resets the screen line link tables.

### Operation:

1. JSR E5A0/E5BB to reset default input device, 9A, to 3 (the screen), and the default output device, 99, to 0 (the keyboard).
2. JMP E566/E581 to home the cursor and reset the screen line link table.

## Display Kernal Messages F12B/F1E2-F13D/F1F4

### Called by:

JMP at F5DA/F672 in Display LOADING/VERIFYING Message; alternate entry at F12F/F1E6 by JSRs at F5B5/F64D and F5BE/F656 in Display SEARCHING FOR Message, JSR at F695/F72E in Display SAVING Filename Message, JSR at F71F/F7A2 in Error Message Handler, JSR at F752/F7D5 in Find Next Tape Header, JSR at F81E/F89B and JMP at F82B/F8A8 in Display Tape Button Messages.

This routine uses the index passed in the Y register to retrieve a message from the Kernal message table at F0BD/F174-F12A/F1E1 and to display this message using CHROUT.

At the entry point F12B/F1E2, it tests 9D, the Kernal message control flag. If the high bit is 0, it doesn't display the messages for loading or verifying.

At the entry point F12F/F1E6, this check of 9D is not made.

### Entry requirements:

The Y register should contain the index into the Kernal message table for the starting character of the desired message.

### Operation:

1. If 9D, the Kernal message control flag, has its high bit cleared to 0, branch to step 10 to CLC and RTS.
2. F12F/F1E6: LDA with character from message table at F0BD/F174 indexed by Y register.
3. Save the accumulator contents from step 2 on the stack.
4. Mask off the most significant bit in the character loaded. This only affects the last character in a message, which has its most significant bit on to indicate end of message.
5. JSR FFD2 (CHROUT) to send this character to the current output channel (defaults to the screen).
6. INY, the index into the message.
7. Restore accumulator value saved in step 3 from the stack.
8. If LDA loaded a byte that has its high bit off, the end of the message has not been reached. Branch to step 2.
9. If LDA loaded a byte that has its high bit on, the end of the message has been reached, continue with step 10.
10. CLC and RTS.

### See If Logical File Exists

**F30F/F3CF-F31E/F3DE**

#### Called by:

JSR at F20E/F2C7 in CHKIN Execution, JSR at F250/F309 CHKOUT Execution, JSR at F351/F411 OPEN Execution; alternate entry at F314/F3D4 by JSR at F291/F34A in Determine Device for CLOSE.

First, clear the I/O status word, 90.

Next, test to see if the logical file number passed in the X register is already in the logical file number table that starts at 0259. If the logical file is in the table, exit with Z set to 1 (BEQ condition). If the logical file is not in the table, exit with Z set to 0 (BNE condition).

### Entry requirements:

The X register should hold the logical file number. Locations 0259-0262 comprise the logical file number table. Location 98 indicates the number of open files.

### **Operation:**

1. Set 90, the I/O status, to 0.
2. TXA, thus accumulator now contains the logical file number.
3. F314/F3D4: LDX 98, the number of open files.
4. DEX. If X is now negative, either there are no open files, or all the open file numbers have been compared and none match the current file number. If X is negative, branch to step 8.
5. Compare the accumulator to the logical file number at 0259, indexed by the X register.
6. If comparison is not equal, branch to step 4. If this branch is taken, the Z flag is 0.
7. If comparison is equal, the Z flag is 1. Fall through to step 8.
8. RTS.

### **Extract Logical File Number, Device Number, and Secondary Address from Tables**

**F31F/F3DF-F32E/F3EE**

#### **Called by:**

JSR at F216/F2CF in CHKIN Execution, JSR at F258/F311 in CHKOUT Execution, JSR at F298/F351 in Determine Device for CLOSE.

The X register value at entry is used as an index to the logical file number table (B8), the device number table (BA), and the secondary address table (B9).

#### **Operation:**

1. Retrieve the logical file number from the logical file number table at 0259 indexed by X register and store this value at B8.
2. Retrieve the device number from the device number table at 0263 indexed by X register and store this value at BA.
3. Retrieve the secondary address from the secondary address table at 026D indexed by X register and store this value at B9.

### **Display LOADING/VERIFYING Message**

**F5D2/F66A-F5DC/F674**

#### **Called by:**

JSR at F5A2/F63A in Control Routine for Tape Load, JMP at

F4F0/E4CC in Load/Verify from Serial Device.

This routine displays either LOADING or VERIFYING depending on whether 93, the LOAD/VERIFY flag is 0 (LOAD) or 1 (VERIFY).

### **Operation:**

1. LDY \$49 to index to LOADING message.
2. If 93, the LOAD/VERIFY flag, is 0, branch to step 4.
3. If 93, the LOAD/VERIFY flag, is 1, LDY \$59 to index to VERIFYING.
4. JMP F12B/F1E2 to display the message.

### **Display SAVING Filename Message**

**F68F/F728-F69A/F733**

#### **Called by:**

JSR at F608/F6A0 in Save to Serial Device, JSR at F669/F702 in Control Routine for Tape SAVE

This routine displays SAVING and the filename for the file being saved.

If 9D, the Kernal message control, has its high bit off, no messages are displayed.

#### **Operation:**

1. If 9D, the Kernal message control, has its high bit off, RTS.
2. LDY \$51 to index to the SAVING message.
3. JSR F12F/F1E6 to display SAVING message.
4. JMP F5C1/F659 to display name of file being saved and RTS.

### **Error Message Handler**

**F6FB/F77E-F72B/F7AE**

#### **Entry Points:**

F6FB/F77E for I/O ERROR #1 (too many open files) by JMP at F35F/F41F in OPEN Execution.

F6FE/F781 for I/O ERROR #2 (file already open) by JMP at F356/F416 in OPEN Execution.

F701/F784 for I/O ERROR #3 (file not open) by JMP at F255/F30E in CHKOUT Execution, JMP at F213/F2CC in CHKIN Execution.

F704/F787 for I/O ERROR #4 (file not found) by JMP at F3AC/F46C OPEN Execution, JMP at F530/F5C7 in Load/Verify from Serial Device.



- F707/F78A for I/O ERROR #5 (device not present) by JMP at F24D/F306 in Open Serial Input Channel, JMP at F28E/F347 in Open Serial Output Channel, JMP at F3F3/F4AF in Prepare Serial Device for Open, Load, or Save.
- F70A/F78D for I/O ERROR #6 (file is not an input file) by JMP at F230/F2E9 in CHKIN Execution, JMP at F34E/F40E in OPEN Execution.
- F70D/F790 for I/O ERROR #7 (file is not an output file) by JMP at F25F/F318 in CHKOUT Execution.
- F710/F793 for I/O ERROR #8 (file name is missing) by JMP at F4BC/F560 in Load/Verify from Serial Device, JMP at F602/F69A in Save to Serial Device.
- F713/F796 for I/O ERROR #9 (illegal device number) by JMP at F390/F450 in Determine If Open Is for Reading or Writing, JMPs at F4A5/F553 and F536/F0B9 in Determine Device for LOAD, JMP at F53E/F5D6 in Control Routine for Tape Load, JMP at F5F1/F689 in Determine Device for SAVE.

This routine is called to handle Kernal error messages. If the Kernal message control flag, 9D, allows error messages, the message I/O ERROR is displayed followed by the number of the I/O error.

This routine is never called by BASIC, which has its own error messages for all the conditions above. BASIC's Display READY routine writes \$80 to location 9D to disable the Kernal error messages.

### **Exit conditions:**

The routine exits with the carry set and the accumulator containing the number of the error message.

### **Operation:**

1. LDA with \$01-\$09, depending on the entry point. See the entry points above for the values loaded for the various conditions.
2. Through the use of BIT instructions, all the other entry points in the routine are bypassed.
3. PHA, storing the I/O error number on the stack.
4. JSR FFCC (CLRCHN) to close serial channels and to reset the default input device to be the keyboard and the default output device to be the screen.

5. If 9D has bit 6 off, branch to step 8.
6. With Y register loaded as 0, JSR F12F/F1E6 to display message I/O ERROR #.
7. Pull the I/O error number from the stack, and then push it back onto the stack again. However, also ORA \$30 to convert it to an ASCII character and JSR FFD2 (CHROUT) to display the I/O error number.
8. PLA, so that accumulator again contains the I/O error number.
9. SEC and RTS.

### **Display SEARCHING FOR Message** **F5AF/F647-F5C0/F658**

#### **Called by:**

JSR at F39E/F45E in Open Logical File for Reading from Tape, JSR at F546/F5DE in Control Routine for Tape Load, JMP at F4C1/E4BE in Load/Verify from Serial Device.

This routine displays SEARCHING. If B7, the number of characters in the filename, is nonzero (indicating a filename exists), then also display FOR and fall through to the following routine which displays the filename. If, however, B7 is zero (indicating no filename exists), display SEARCHING and exit.

Also, if 9D, the Kernal message control flag, has its high bit off, no messages are displayed.

#### **Operation:**

1. If 9D, the Kernal message control, has its high bit off, RTS.
2. LDY \$0C as an index to SEARCHING message and JSR F12F/F1E6 to display the message.
3. If B7, the number of characters in the filename, is 0, RTS.
4. If B7 is nonzero, indicating a filename exists, LDY \$17 as an index to the FOR message and JSR F12F/F1E6 to display the message.
5. Fall through to F5C1/F659 to display the filename.

### **Display Filename** **F5C1/F659-F5D1/F669**

#### **Called by:**

JMP at F698/F731 in Display SAVING Filename Message, fall through from F5BE/F656 in Display SEARCHING FOR Message.

If a filename exists, this routine displays the filename that is part of a SEARCHING FOR or SAVING message.

**Operation:**

1. LDY B7, the number of characters in the filename.
2. If B7 contains zero, indicating no characters in the filename, branch to step 9 to RTS.
3. LDY \$00.
4. LDA with the next character in the filename; (BB) points to the start of the filename, and the Y register is used as an index through the filename.
5. JSR FFD2 (CHROUT) to display the character from the filename.
6. INY.
7. CPY to B7, the number of characters in the filename.
8. If not equal, branch to step 4; if equal, fall through to step 9, as all characters from the filename have been displayed.
9. RTS.



## **Chapter 7**

---

# **Screen Routines**



# Screen Routines

---

## Screen Routines Overview

The Kernal screen editor routines are also used by BASIC to fill the BASIC input buffer at 0200. The screen editor routines allow you to use the cursor control keys to move the cursor to any line on the screen and edit that line.

Two screen-related routines that are not used directly by BASIC are the routine to return the number of columns and rows in the screen and the routine to set or read the cursor location. The latter routine to read the cursor location is called by the BASIC POS command. Although the tape, RS-232, and serial routines do not directly call these screen editor routines, the Main Screen Editor routine is called from CHROUT, and the CHRIN from Keyboard or Screen routine is called by CHRIN.

This section covers the process of typing a line on the screen (with editing characters explained), displaying this line on the screen, and storing the line in the buffer at 0200 when the RETURN key is pressed. The descriptions in this section do not explain how to use bitmapped graphics on the screen.

Although the normal screen size on the VIC is 22 columns by 23 lines and on the 64, 40 columns by 25 lines, the screen editor routines for the two computers are the same. The differences in the routines are:

- Size of the screen line link table—24 entries on the VIC, 26 on the 64.
- Number of physical lines per logical line—1-4 on the VIC, 1-2 on the 64.
- Maximum number of characters per logical line—88 on the VIC, 80 on the 64.
- Addition of 22 characters per physical line (VIC) or 40 characters per physical line (64) when using the routines that add or subtract physical line length (such as the Advance Cursor to Next Screen Line routine).

Two kinds of lines are used by the screen editor: a physical line is one line of 40 (64) or 22 (VIC) characters that physically extends across the screen; a logical line is made up of 1-2/1-4 physical lines. The screen line link table contains one

byte that corresponds to each physical screen line. The table for the VIC has 24 entries and a twenty-fifth which holds \$FF to distinguish bottom of screen, while the 64 has 26 entries and a twenty-seventh holding \$FF to indicate the bottom of the screen. Although the VIC normally uses only 23 lines, you can change the display registers to allow 24 lines. Any physical screen line that has the high bit on in its link table entry denotes a physical line that is the start of a logical line. Physical screen lines that have the high bit off in the link table entry are continuations of a logical line. For the VIC, there can be up to three continued physical screen lines for a logical line, while for the 64 there can be only one continued line.

The screen line link table bytes also indicate in which page of memory the screen memory is located. The screen is a memory-mapped I/O device, which means that writing to a screen memory location directly changes the screen display, and reading from a screen memory location shows exactly what is at that location on the screen.

You can easily determine the location of any character on the screen. The link byte for that line is used to determine the page (or high byte) of memory in which the screen is located. Another table contains the low bytes of the screen memory addresses. This latter table returns the low byte which is combined with the high byte to determine the start of the line in screen memory. The column within the logical screen line is then used to retrieve a particular screen address (which typically is the location of the cursor on the screen).

The screen editor keeps track of which line the cursor is on and whether this physical line is a complete logical line or just one physical line in a logical line. When BASIC detects a RETURN, the logical line is placed in the BASIC input buffer at 0200. This logical line can be either a direct mode or a program mode line. You edit logical lines, not physical lines.

Although the screen editor is designed for BASIC, you could use the screen editor routines in your own machine language programs by treating the screen in the same manner as BASIC does. You are limited to the 40 column by 25 line/22 column by 23 line size screen if you use the screen editor without modification. See the BASIC routine at A560/C560 for an example of how to set up a machine language routine to use the screen editor routines to fill an input buffer. If you



## Screen Routines

---

were implementing another language on the VIC, you could make use of the existing screen editor to fill the buffer at 0200 and then process the buffer when RETURN is pressed.

The VIC and 64 *Programmer's Reference Guides* cover the control keys and how to edit the screen so there's no need to repeat that information here. See the detailed descriptions in the Main Screen Editor routine for comments about how each key actually functions.

The 64/VIC has a full screen editor, which allows changes at any location on the screen, rather than just on one particular line as did some old-fashioned line editors.

An interesting project would be to write a program to convert the VIC to a 40 column by 25 line screen editor or the 64 to an 80 column by 25 line screen editor all under software control. In fact, some products for the 64 do provide for an 80-column screen through software routines.

Because of the limitation of the number of pixels on the VIC, it is impossible to have an 80 column screen without making hardware modifications or adding cartridges with hardware to handle an 80-column screen. (On the VIC, 22 columns by 8 pixels per column = 176 total horizontal pixels. If you tried to convert to 80 columns that would leave only 2.2 pixels per character—not enough with which to form characters.) However, the 40 column screen is possible ( $176/40 = 4.4$  pixels per character in a 40 column screen). On the 64, software simulation of 80 columns is possible since with 320 pixels per line you have 4 pixels per character.

In some older 64s (version 2 of the Kernal), a bug in the screen editor can hang up your system on occasion. For example, when you enter a full logical line at the bottom of the screen, the line scrolls, and you hit the delete key, the system locks up. When this occurs, the CIA registers are overwritten by the screen editor as it writes past the end of color RAM that immediately precedes the CIA registers. Program 1-1 demonstrates one solution to the lockup problem. Also, refer to *COMPUTE!'s Gazette*, May 1984, page 108, for a couple of other fixes to the lockup problem. This bug has been corrected in version 3 of the Kernal, found in more recent 64s.

The screen editor routines make use of the following variables:

## Screen Routines

### Screen Variables

Hex Location	Description
99	Default input device
9A	Default output device
(AC)	Temporary pointer for moving screen lines
(AE)	Temporary pointer for moving color lines
C7	Flag for reverse-video characters
C8	Length of logical line for input
C9	Logical line number for input
CA	Logical column number for input
CC	Cursor blink switch
CD	Cursor blink countdown
CE	Character under cursor
CF	Cursor blink status
D0	Keyboard/screen input switch
(D1)	Pointer to current logical line
D3	Position of cursor in logical line
D4	Flag for quote mode
D5	Length of current Logical line
D6	Physical line number
D7	Character to be displayed
D8	Number of inserts outstanding
D9-F2/F1	Screen line link table
(F3)	Pointer to current color memory line
0286	Current character color
0287	Color under cursor
0288	Current screen memory page
0291	Character set switch
0292	Downward scroll switch

### Clear Screen Line Cursor Is On E9FF/EA8D-EA12/EAA0

#### Called by:

JSR at E560/E57B in Set VIC Chip Registers, Clear Screen, Home Cursor, Set Screen Line Link Table, JSR at E9A6/EA2C in Insert Blank Line ; JSR at E913/E99D in Scroll Screen.

For all 40/22 columns of the current physical line, this routine puts spaces into every position and sets each of these spaces to the color indicated by background color register 0 (64 with version 2 of the Kernal), to the current foreground color indicated by 0286 (64 with version 3 of the Kernal), or to white (VIC). In effect, the current screen line is blanked out.

### Entry requirements:

The X register should hold the number (0–24/22) of the physical screen line to be cleared. The value will be used as an index into the ROM screen line link table for the low byte of the starting address of the line to be retrieved, and into the RAM screen line link table in page zero for the high byte of the address.

### Exit conditions:

The current screen line is blanked. (D1) is the pointer to the start of the screen line the cursor is on. (F3) is the pointer to the color nybble corresponding to the starting location of the current screen line.

### Operation:

1. JSR E9F0/EA7E to set (D1), the pointer to the start of the screen line the cursor is on.
2. JSR EA24/EAB2 to set (F3), the pointer to the color nybble corresponding to the start of the current screen line.
3. 64: For all 40 columns of the current physical screen line, store a space of the color indicated by background color register 0 (version 2 of the Kernal) or to the current foreground color indicated in 0286 (version 3 of the Kernal).

VIC: For all 22 columns in the current physical screen line, store a white space.

## Set Pointers to Screen Line Cursor Is On E9F0/EA7E–E9FE/EA8C

### Called by:

JSR at E990/EA17 in Insert Blank Line, JSR at EA01/EA8F in Clear Screen Line Cursor Is On, JSR at E900/E98A in Scroll Screen, JMP at E6F4/E720 in Advance Cursor and Scroll or Insert Blank Lines.

The value in the X register at entry is used as an index into the screen line link table in ROM for the low byte of the pointer to the start of the screen line the cursor is on.

The X register value also serves as an index into the screen line link table at D9, used to set the high byte of the pointer to the start of the screen line the cursor is on.

As an example of how this routine functions (on the VIC), consider what happens if the value in the X register at entry is 2 and if the screen memory starts at 1000 (as is normally the

case when 8K or more expansion memory is installed). The low byte retrieved from the ROM table at  $EDFD + X = EDFF$  will be \$2C (44 decimal), which is saved in D1, the low byte of the pointer to the current screen line.

Then a value is retrieved from the page zero screen line link table at  $D9 + X = DB$ , which in this case will be \$90. That value is ANDed with \$03 to mask off all but the two low bits, resulting in zero. This result is ORed with the contents of 0288, the screen memory page, which will contain \$10. The end result is stored in the high byte of the pointer to the current screen line, D2. Thus, (D1) holds \$102C, which points to the start of the third line on the screen.

You could use this routine in a machine language game program, for example, to determine the start of the screen memory line the cursor is on after calling the Kernal routine PLOT to determine the current row of the cursor. PLOT will put the row number into the X register, as required by this routine.

### Entry requirements:

Location 0288 must contain the value of the current screen memory page number (default of \$04/\$1E or \$10). The X register contents will determine the index into the screen line link table and the ROM table of low byte values for the pointer into screen memory.

### Exit conditions:

(D1) points to the start of the screen line the cursor is on.

### Operation:

1. Get the low byte of the offset into screen memory from the table at  $ECF0/EDFD$ , indexed by the X register.
2. Save as the low byte of the pointer to the start of the line the cursor is on, D1.
3. Get a byte from the screen line link table at D9, indexed by the X register.
4. AND with \$03 to mask out all but the two lowest bits.
5. OR with the contents of 0288, the screen memory page.
6. Save as the high byte of the pointer to the start of the screen line the cursor is on, D2.

### **Set VIC Chip Registers, Clear Screen, Home Cursor, Set Screen Line Link Table E518-E599/E5B4**

#### **Called by:**

JSR at FF5B in Initialize VIC-II Chip and Set PAL/NTSC Flag (64), JSR at FD38 in System Reset (VIC), JSR at FE6C/FED8 in BRK Interrupt Handler; alternate entry at E544/E55F (Clear Screen) by JSR at E86F/E8B5 in Main Screen Editor; alternate entry at E566/E581 (Home Cursor) by JSR at E790/E7BB in Main Screen Editor, JSR at E59E/E5B8 in Set I/O Defaults and Home Cursor; alternate entry at E56C/E587 (Set Screen Line Link Table) by JSR at E511/E510 in Read/Plot Cursor Location, JSR at E70F/E73A in Move Cursor to Previous Screen Line, JSRs at E848/E88E and E88F/E8D5 in Advance Cursor to Next Screen Line.

Set the screen as the output device and the keyboard as the input device, and initialize VIC chip registers.

**VIC:** The VIC chip registers that specify the screen memory location (bit 7 of 9002 and bits 4-6 of 9005) are set based on the value in the screen memory page, 0288, which was set when memory was initialized.

**64:** 0288 is initialized to \$04 during system initialization to set the default screen memory location to start at 0400.

The flag to allow character set switching by holding down the SHIFT and Commodore keys together is enabled, the cursor blink flag is set to indicate that the character under the cursor is not reversed, and the pointer to the Keyboard Table Setup routine, (028F), is set to EB48/EBDC. The maximum number of characters in the keyboard buffer is set to \$0A (10 decimal). The delay before the first repeat of a key is also set to \$0A. The current character color is initialized to \$0E (light blue)/\$06 (blue). The delay before repeats of a key following the first repeat is set to \$06. The count before blink of the cursor is set to \$0C (12 decimal).

*Clear Screen entry point:* For the VIC, the routine initializes the screen line link table values to \$9E or \$9F if screen memory page is \$1E, or to \$90 or \$91 if screen memory page is \$10. For the 64 with the default screen at 0400, the values are \$84 and \$85. All 25/23 lines of the screen are then blanked out, working up from the bottom of the screen.

*Home Cursor entry point:* Set the cursor position for current physical screen line to 0 and the cursor position within the line to 0.

*Set Screen Line Link Pointer entry point:* Set the position of the cursor within this logical screen line, D3, by adding 40/22 to the initial cursor location within the physical screen line until all previous physical lines for this logical line have been scanned. Set the pointer to the start of the logical screen line the cursor is on, (D1), based on which screen line (current physical screen line or any previous) has its high bit on in that line's screen line link table entry. A high bit on in a line link table entry indicates the corresponding screen line is a starting line, while a high bit off indicates the corresponding screen line is a continued line. If no lines have the high bit on in the table entry, set the pointer to the start of the screen line to the address of the start of screen memory (0400/1E00 or 1000). Find the start of the current logical screen line, then scan down the screen for the first line that has its screen line link table entry with the most significant bit on. Each line scanned in this downward search adds 40/22 to the initial value of 39/21; this value is then stored as the maximum logical length of this line.

If you change the location of screen memory, you must notify the screen editor by changing 0288 (648 decimal) to the starting page of screen memory.

Possible values of screen line link tables for the two normal screen memory locations are:

On the 64, screen memory at 0400: \$84, \$85, \$86, or \$87 if the physical line is the start of a logical line, \$04, \$05, \$06, or \$07 if the physical line is the continuation of a logical line.

On the VIC, screen memory at 1E00 (unexpanded VIC): \$9E or \$9F if the physical line is the start of a logical line, \$1E or \$1F if the physical line is the continuation of a logical line.

On the VIC, screen memory at 1000 (VIC with 8K or more expansion): \$90 or \$91 if the physical line is the start of a logical line, \$10 or \$11 if the physical line is the continuation of a logical line.

### Operation:

1. JSR to E5A0/E5BB to set input device to keyboard, output device to screen, and initialize VIC-II/VIC chip registers.
2. VIC only: Load screen memory page from 0288. Mask out

## Screen Routines

bit 1, then ASL twice (to multiply the value by 4). Then turn on bit 7 and store this value in 9005 as bits 10–13 of the screen address. Also, by clearing the low nybble that is stored into 9005, the character set at 8000 (uppercase/graphics) is selected for the screen characters. If screen memory page bit 1 is on (for example, if the page value is \$1E), set the high bit of 9002, which is bit 9 of the address of the screen, to 1.

As an example, consider the calculations for screen memory at 1E00, or page \$1E.

\$1E =	0001 1110
Mask bit 1	<u>1111 1101</u>
	0001 1100
ASL	
ASL	
	0111 0000
ORA \$80	<u>1000 0000</u>
	1111 0000

Bits 4–7 now become bits 10–13 of the screen memory address. Bit 9 of the screen memory address is taken from the high bit of 9002, which has been set to 1 because the screen memory page, 0288, has its bit number 1 on. Bit 13 of the screen address must be 1.

Thus, the address generated is:

<b>Bit:</b>	13	12	11	10	9	8	7	6	5	4	3	2	1	0
<b>Value:</b>	1	1	1	1	1	0	0	0	0	0	0	0	0	0

Bits 14 and 15 are not used in decoding the address to retrieve contents of character ROM, screen RAM, or color ROM, as only 13 address lines are connected to these chips.

3. Store \$00 in 0291 to enable the flag for character set switching. When enabled, this flag allows holding the Commodore and SHIFT keys down together to switch to the alternate character set.
4. Set cursor blink flag at CF to 0, indicating nonreversed character under cursor.
5. Set the pointer to Keyboard Table Setup routine at (028F) to EB48/EBDC.
6. Set the maximum number of characters in keyboard buffer, 0289, to \$0A (10 decimal).

7. Set the delay before the initial repeat of a key, 028C, to \$0A also.
8. Set the current character color, 0286, to \$0E (light blue)/\$06 (blue).
9. Set the delay before following repeats of a key, 028B, to \$04.
10. Set the count before blink of cursor, CD, to \$0C (12 decimal). Also store the \$0C in CC, which turns off the cursor since any nonzero value in CC disables cursor blinking.
11. E544/E55F (Clear Screen entry point): Set screen line link table values at D9-F2/F1 based on 0288, the screen memory page. For the unexpanded VIC, where the screen memory page is \$1E, D9-E4 = \$9E and E5-F0 = \$9F. For the VIC with 8K or more expansion, where the screen memory page is \$10, D9-E4 = \$90 and E5-F0 = \$91. For the 64, where the screen memory page is \$04, , D9-DF = \$84, E0-E5 = \$85, E6-EC = \$86, and ED-F2 = \$87. The routine then sets the byte following the table, F3/F1, to \$FF.

The maximum number of table entries for the VIC is 25, including the ending \$FF. For the 64, the maximum number of entries is 27, including the ending \$FF. However, the ending \$FF serves no purpose in the 64, and it is overwritten by screen routines that use (F3) as a pointer into color memory.

12. LDX with the value for the bottom line of the screen, then JSR to E9FF/EA8D (Clear Screen Line Cursor Is On) blank out the current line. Next, DEX and loop to the JSR to blank out the current line until all 25/23 lines of the screen are erased.
13. E566/E581, Home Cursor entry point: Set D3, cursor column, and D6, current physical screen line cursor is on, to 0. Column 0 of line 0 is the upper left corner of the screen, the home position.
14. E56C/E587, Set Screen Line Link Table entry point: LDX with the value from D6, the physical screen line the cursor is on LDA with the value from D3, the column in the current physical screen line the cursor is on.
15. LDY with an entry from the screen line link table at D9, using the line value in X as an index. If the high bit of this entry is on, branch to step 18, as this line is not a continuation of the previous line.



16. Add \$28/\$16 (40/22 decimal) to D3, the logical cursor position within the line.
17. DEX. If  $X \geq 0$ , branch to step 15. If  $X < 0$  (i.e., if X was 0 before the DEX, indicating that top line has been reached), fall through to step 18. This step prevents the routine from attempting to search beyond the start of the table.
18. LDA with the entry from the screen line link table at D9 indexed by X; AND with \$03 to mask out all but bits 0 and 1, then ORA with the contents of 0288, the screen memory page. The resulting value is the high byte of the address of the start of the current logical screen line.
19. STA in D2, the high byte of the pointer to the start of this logical screen line.
20. Set the low byte of the pointer to the start of this logical screen line, D1, from the table of low byte offsets at ECF0/EDFD, indexed by X. (Version 3 of the 64 Kernal accomplishes steps 18–20 with a JSR E9F0, which performs the same steps).
21. Now search downward in the table for the end of the current logical screen line, which is indicated by a line link table entry with its high bit on. Each line scanned in this downward search adds \$28/\$16 (40/22 decimal) to the initial current screen line length of \$27/\$15 (39/21 decimal). The final total is then stored in D5, the length of the current logical line.
22. (Version 3 of 64 Kernal ROM only): JMP EA24 to set (F3) to point to the color memory location corresponding to the start of the logical screen line (D1).

### Set Default Device Numbers

#### E5A0/E5BB-E5A7/E5C2

##### Called by:

JSR at E518 Set VIC Chip Registers, Clear Screen, Home Cursor, Set Screen Line Link Table, JSR at E59A/E5B5 in Set I/O Defaults and Home Cursor.

The default device for output, 9A, is set to \$03 to make the screen the current output device; the default device for input, 99, is set to \$00 to make the keyboard the current input device. The routine then falls through to the following routine to initialize VIC chip registers.

## Operation:

1. Set 9A, the output device number to \$03.
2. Set 99, the input device number to \$00.
3. Exit by falling through to following routine, E5A8/E5C3 to initialize VIC chip registers.

## Initialize VIC Chip Registers E5A8/E5C3-E5B3/E5CE

### Called by:

Falls through from E5A7/E5C2 in Set Default Device Numbers, JSR at FDEB in Initialize Memory Pointers (VIC).

The VIC chip registers for the VIC are loaded from values in a table at EDE4; the VIC-II chip registers are loaded from table values at ECB9 for the 64.

### Operation:

VIC: For each of the 16 values in the table, load the table value and store this value into the corresponding VIC chip register at 9000-900F. See the table below for the VIC register settings.

#### VIC 6560-6561 Initial Register Settings

9000	0000	0101	Interlaced mode off; horizontal screen origin = 5
9001	0001	1001	Vertical screen origin = \$19 (25 decimal)
9002	1001	0110	Screen address VA9 = 1; 22 columns
9003	0010	1110	Raster value = 0; 23 video rows; 8 × 8 character size
9004	0000	0000	Raster value = 0
9005	1011	0000	Screen address VA13-10 = 1011; character address VA13-10 = 0000
9006	0000	0000	Light pen horizontal position
9007	0000	0000	Light pen vertical position
9008	0000	0000	Paddle X digitized value
9009	0000	0000	Paddle Y digitized value
900A	0000	0000	Bass voice off
900B	0000	0000	Alto voice off
900C	0000	0000	Soprano voice off
900D	0000	0000	Noise source off
900E	0000	0000	Audio volume off; auxiliary color black
900F	0001	1011	White background; normal (nonreversed) characters; cyan border

64: For each of the 47 values in the table, load the table value and store this value into the corresponding VIC-II chip

## Screen Routines

register at D000–D02E. See the table below for the 64 register settings.

### VIC-II 6567 Register Initial Settings

D000–D00F	0000	0000	X and Y coordinates = 0 for all sprites
D010	0000	0000	MSB of X coordinate = 0 for all sprites
D011	1001	1011	Raster register bit 8; disable extended color mode; disable bit map; don't blank screen; 25 screen rows; smooth scroll to Y dot position 3
D012	0011	0111	Raster register bits 7–0 (raster register 8–0 = 1 0011 0111 = \$0137, 311 decimal)
D013	0000	0000	Light pen horizontal position
D014	0000	0000	Light pen vertical position
D015	0000	0000	Disable all sprites
D016	0000	1000	Reset = 0 for normal display; disable multicolor mode; 40 column screen; smooth scroll to X dot position 0
D017	0000	0000	No vertical expansion for any sprite
D018	0001	0100	Video matrix VA13–VA10 = 0001; character set VA13–VA11 = 010
D019	0000	1111	Clear any pending VIC chip interrupts
D01A	0000	0000	Disable VIC chip interrupts
D01B	0000	0000	All sprites have display priority over background
D01C	0000	0000	Disable multicolor mode for all sprites
D01D	0000	0000	No horizontal expansion for any sprite
D01E	0000	0000	Sprite to sprite collision detection register
D01F	0000	0000	Sprite to background collision detection register
D020	0000	1110	Border color = light blue
D021	0000	0110	Background color 0 = blue
D022	0000	0001	Background color 1 = white
D023	0000	0010	Background color 2 = red
D024	0000	0011	Background color 3 = cyan
D025	0000	0100	Sprite multicolor 0 = purple
D026	0000	0000	Sprite multicolor 1 = black
D027	0000	0001	Sprite 0 color = white
D028	0000	0010	Sprite 1 color = red
D029	0000	0011	Sprite 2 color = cyan
D02A	0000	0100	Sprite 3 color = purple
D02B	0000	0101	Sprite 4 color = green
D02C	0000	0110	Sprite 5 color = blue
D02D	0000	0111	Sprite 6 color = yellow
D02E	0100	1100	Sprite 7 color = medium gray

### **CHRIN from Keyboard or Screen** **F157/F20E-F172/F229**

#### **Called by:**

Indirect JMP through (0324) from Kernal CHRIN Vector at FFCF; alternate entry at F166/F21D by BNE at F144/F203 in GETIN Preparation.

If the current input device is the keyboard, set CA, the current physical line, and C9, the current cursor location, from D6 and D3, respectively. Then JMP to E632/E64F to input characters until a carriage return is encountered.

If the current input device is the screen, set D0 to 3 to indicate input from screen and set C8, the pointer to the end of line for input, from D5, the maximum logical length of this line. Then JMP E632/E64F to input characters from the screen until a carriage return is detected.

#### **Entry requirements:**

Location 99 should hold the number of the current input device. Location D3 should hold the cursor position within the current logical line and D6 holds the current physical screen line the cursor is on. Location D5 holds the length of the current logical line (21, 43, 65, or 87 for the VIC; 39 or 79 for the 64).

BASIC calls CHRIN to fill the BASIC input buffer at 0200.

#### **Operation:**

1. If 99, the current input device, is nonzero, branch to step 5. If the input device is 0 (keyboard), fall through to step 2.
2. Set C9 to the value in D3, which is the cursor position within the current logical line.
3. Set CA to the value in D6, the current physical screen line the cursor is on.
4. JMP E632/E64F to retrieve characters from the keyboard buffer and display them on the screen until a carriage return is retrieved from the buffer. Exit with the accumulator containing the ASCII value of the carriage return or of the last key pressed.
5. For CHRIN from the screen, first set D0, a flag indicating whether input is from the screen or keyboard (3 represents screen input).
6. Set C8, pointer to end of line for input, to 39 or 79 for the 64 or to 21, 43, 65, or 87 for the VIC, based on the value in D5, the length of the current logical line.

7. JMP E632/E64F to retrieve characters from screen until a carriage return is retrieved. Exit with the accumulator containing the ASCII value of the carriage return or the character under the cursor.

### **Get Character from Keyboard or Screen E632/E64F-E683/E6B7**

#### **Called by:**

JMPs at F163/F21A and F170/F227 in CHRIN from Keyboard or Screen; alternate entry at E63A/E657 by BMI at E61D/E638, BNE at E626/E641, and BCC at E62E/E64B in Get Characters Until RETURN Key Detected; alternate entry at E65D/E691 by BCS at E630/E64D in Get Characters Until RETURN Key Detected.

CHRIN calls this routine to retrieve a character from the current input device. Characters that are typed on the keyboard are displayed on the screen, and the screen display codes are converted back to ASCII characters to be returned to CHRIN.

This somewhat odd sequence of converting the keyboard entries to screen codes, displaying them on the screen, reading the screen line and converting screen codes to ASCII equivalents is there for a good reason. The sequence allows you to hit RETURN at any location on the current logical screen line and have the entire logical line placed into the BASIC input buffer. Changes made anywhere on the line are picked up no matter where the cursor is within the logical line when RETURN is entered.

If the input device as indicated by D0 is the keyboard, branch to E5CD/E5E8 to retrieve characters from the keyboard queue until RETURN is entered. Return to this routine from the keyboard routine either at E63A/E657 or E65D/E691. Once the keyboard routine retrieves a RETURN character from the keyboard buffer, it considers the logical line complete and resets D0, the flag for input from screen or keyboard, to a nonzero value, returning to this routine at E63A/E657 or E65D/E691. The next time this routine is called to return a character for CHRIN, it finds the input device flag D0 has been set to this nonzero value, and thus just falls through to input only from the screen. Following calls repeat this process until finally the end of the screen line is reached, at which time the input flag D0 is reset to 0 to allow input from the keyboard.

If the input device is the screen, just retrieve the character from the current logical screen line indexed by the column the cursor is on within the logical line. The routine also retrieves a character from the current screen line upon returning from the call to the keyboard routine. This return occurs when a RETURN has been entered when either this screen line is a continuation line or the cursor for this screen line is located before the end of the line.

Convert this screen code to the corresponding ASCII code. If the column this screen code is taken from is equal to the end of the line, reset D0 (input from keyboard/screen) to 0. If the input device is the screen JSR E716/E742 to echo the character to the screen.

If the column from which this screen code is taken is equal to the end of the line, set the ASCII value of the last key pressed to the value for the RETURN key.

If the column from which this screen code is taken is not equal to the end of the line, store the converted screen code into the ASCII character for the key last pressed at D7.

Finally, if the screen code is \$DE, which is the screen code for the symbol  $\pi$ , set the ASCII value of the last key pressed to \$FF. Three different ASCII values represent the  $\pi$  key.

The actual character code used is Commodore's eight-bit version of ASCII, rather than the industry-standard seven-bit code.

To convert the screen code to the equivalent ASCII value, this routine performs the manipulations shown below:

### Screen Code to CBM ASCII Conversion

Screen code (decimal)	Modification
-----------------------	--------------

0-31 or 128-159	ORA \$40 (no change if within quotation marks)
32-63 or 160-191	No change
64-127 or 192-255	ORA \$80

If you design your own routine to convert screen codes to ASCII codes, you could use the instructions from E63E/E672 through E656/E686 as a model. Copy these instructions to your own routine for screen-code-to-ASCII conversion, and place the screen code value in the accumulator before jumping into the routine. The equivalent ASCII code will be returned in the accumulator.

### Entry requirements:

Location will determine D0 whether input is from keyboard (0) or screen (nonzero). D3 should indicate the cursor position within the logical screen line. (D1) should point to the start of the logical screen line the cursor is on. D4 will indicate if the characters being retrieved are within quotes: 0 if not in quote mode, 1 if in quote mode. C8 should contain the length of the logical line.

### Exit conditions:

D7 holds the ASCII value of the last key pressed. If the current input device is the keyboard, D7 should return with a value of \$0D for RETURN key. If the current input device is the screen, D7 returns with value of either \$0D for RETURN key or the ASCII equivalent of the screen code currently under the cursor.

D3, the cursor position within the logical screen line, is incremented if the cursor was not already on the end of the line.

D0 indicates whether input is from the keyboard (0) or screen (3). It is set to 0 if the end of the current logical line is reached, as indicated by a RETURN key ASCII value being returned in D7.

The carry flag is clear on exit.

### Operation:

1. Save Y and X registers on stack.
2. If D0 is 0, indicating input from keyboard, branch to E5CD/E5E8 (see Get Characters from Keyboard Until RETURN Key Detected routine) to retrieve characters from keyboard queue until RETURN is entered. Program execution eventually returns to this routine at E63A/E657, step 4, or E65D/E691, step 12.
3. If D0 is nonzero, input is from the screen, so fall through to step 4.
4. E63A/E657: LDY from D3, the cursor position within this logical screen line. (Possible values of 0-79/0-87.)
5. LDA (D1),Y. (D1) points to the start of this logical line in screen memory; add the cursor position within this logical line; load accumulator with the screen code of the character under the cursor.
6. Store this screen code in D7.

7. Convert the screen code to an ASCII equivalent. Bit 7 of the original screen code is ignored in this conversion. Thus, reversed screen characters are converted to the same ASCII values as the corresponding unreversed screen code.

First, AND \$3F to set bits 6 and 7 of ASCII code being built to 0.

Next, if bit 6 of the original screen code = 1, ORA \$80. Thus, for screen codes of  $x1xx\ xxxx$  (64–127 and 192–255), the equivalent ASCII code is obtained by ORA \$80.

If quote mode is on, indicated by a nonzero value in D4, branch around the next step of ORA \$40.

If quote mode is off and the original bit 5 of the screen code = 0, ORA \$40. This catches codes with the bit pattern  $x10x\ xxxx$  (codes from 64–95 or 192–223) or  $x00x\ xxxx$  (codes from 0–31 or 128–159).

If quote mode is on, the screen codes from 0–31 or 128–159 are left as is, thus allowing the control characters within quotes to be converted to ASCII.

If bit 6 was 0 and bit 5 was 1 (pattern of  $x01x\ xxxx$  for screen codes from 32–63 or 160–191), leave the screen code unchanged as the screen codes for these ranges are the same code as the equivalent ASCII code.

As an example, consider the screen code number for Z, which is \$1A = 0001 1010. AND \$3F does not affect this code because the two highest bits are already 0. Because bit 6 is 0, we don't ORA \$80. Because bit 5 is 0, we ORA \$40 to set it to 0101 1010, \$5A, the ASCII code for Z.

Another example is the conversion process for changing the representation on the screen for the WHT (CTRL-2) key within quotes, which is the reversed E, to the ASCII value of 5, which is interpreted when sent to the screen as a command to change the character color to white. The screen code for a reversed E is 1000 0101. AND \$3F changes this to 0000 0101. Since bit 6 is 0, the ORA \$80 is not done. Since this reversed E would appear within quotes (quote mode flag on), no ORA \$40 is done. Thus, the ASCII value is 0000 0101 or 5.

8. Increment the cursor location within the logical line, D3, since this routine returns a character from the position under the cursor.



## Screen Routines

---

9. JSR E684/E6B8 to see if the character under the cursor was the quote character. If so, flip the value in the quote flag, D4.
10. CPY C8. The Y register contains the column the cursor was on in the logical screen line. If the input device is the screen, C8 was set from D5, the current logical line length (64: 39 or 79; VIC: 21, 43, 65, or 87). If the input device is the keyboard, C8 will be set to the value of the length of the current logical line, D5, minus the number of spaces at the end of the line + 1.
11. If the above comparison evaluates as equal, the end of the screen line has been reached, so fall through to step 12. If the above result is not equal, the end of the screen line has not yet been reached, so branch to step 17.
12. E65D/E691: Once the end of the screen line is reached, reset D0, the flag indicating whether input is from screen or keyboard, to 0 for keyboard input.
13. If 99, the device number of the current input device, contains \$03 (screen input), branch to step 15.
14. If 9A, the device number of the current output device, contains \$03 (screen output), branch to step 16.
15. JSR E716/E742 to echo this character on the screen if current input is from the screen.
16. LDA \$0D, ASCII value for the RETURN key. If the end of the line has been reached, the line is followed by a RETURN.
17. STA D7, ASCII value of the last key pressed. Either store the \$0D for RETURN if falling through from step 16, or store the ASCII character for the screen code under the cursor if branching here from step 11, when the end of the line has not yet been reached.
18. Restore the X and Y registers from stack.
19. Load accumulator with the value from D7.
20. If accumulator is now equal to \$DE, which is an ASCII code for  $\pi$ , substitute \$FF, another ASCII code for  $\pi$ , in the accumulator.
21. CLC and RTS.

## Get Characters Until RETURN Key Detected E5CA/E5E5-E631/E64E

### Called by:

Normal entry point is E5CD/E5E8 by BEQ at E638/E655 in Get Character from Keyboard or Screen.

This routine handles CHRIN from the keyboard, looping until characters are in the keyboard queue, and testing each character received to see if it is the RETURN key. If it is not the RETURN key, display the character on the screen. If it is the RETURN key, the keyboard entry or editing is complete.

These loops for characters in the keyboard buffer and for the RETURN key are the reason that CHRIN can loop infinitely.

When the current line is complete, the routine determines the actual number of characters in the current line. This index to the end of the characters + 1 is stored in C8. Thus, C8 contains the number of characters in the line (including the RETURN). Location D0 is reset to indicate input from screen, screen scrolling is enabled, the cursor column is set to 0, and the quote mode flag is turned off. Then, if the cursor is not at the end of the logical screen line, it branches to E63A/E657 to read from the screen. The branch to E63A/E657 is also taken if the current physical line is different than it was when this routine was called, most likely being a continuation line. If the cursor is at the end of the logical screen line, the routine branches to E65D/E691 to reset D0 to input from keyboard, and considers the current line complete.

If the left SHIFT and RUN/STOP keys are held down together, the routine puts LOAD [RETURN] RUN [RETURN] in the keyboard buffer.

The SHIFTed RETURN is not the same as the RETURN without SHIFT as far as this routine is concerned. SHIFTed RETURN does not close the logical screen line as the unSHIFTed RETURN does.

### Operation:

1. LDA C6, the number of characters in the keyboard buffer.
2. STA in CC and 0292. Thus, when no characters are in the keyboard queue, cursor blinking and screen scrolling are enabled. When characters are in the keyboard queue, cursor blinking flag is disabled as is screen scrolling.
3. If there are no characters in the keyboard queue, branch to

## Screen Routines

---

- step 1. Fall through to step 4 when there are characters in the keyboard queue.
4. Disable IRQ interrupts.
  5. If CF, cursor blink status, is nonzero, indicating the character under the cursor is reversed, restore the original character under the cursor, restore its color, reset the cursor blink status to zero, and store the un-reversed character to the screen.
  6. JSR E5B4/E5CF to retrieve the next character from the keyboard queue. This routine also enables IRQ interrupts.
  7. If the ASCII value retrieved from the keyboard queue is \$83 (when the left SHIFT and RUN/STOP keys are held down together), put the characters LOAD [RETURN] RUN [RETURN] (where [RETURN] indicates \$0D, the ASCII character for RETURN) into the keyboard queue. The characters are taken from a table at ECE7/EDF4. Then branch to step 1.
  8. If the ASCII value retrieved was not \$0D (RETURN key), branch to E5CA/E5E5. At E5CA/E5E5, JSR E716/E742 to the Main Screen Editor routine to display the ASCII character retrieved from the keyboard queue on the screen. Upon return from this JSR, continue with step 1 of this routine.
  9. If the ASCII value retrieved was \$0D (RETURN key), continue with step 10. Thus keyboard entry of RETURN does not cause a display on the screen; rather, it causes the end of editing for this logical line.
  10. LDY D5, the current screen line logical length (21, 43, 65, or 87 for VIC; 39 or 79 for 64).
  11. STY D0. By setting D0 to this nonzero value, the next time the routine to get a character from screen or keyboard is called, the nonzero value forces the routine to get the character from the screen, once the entire keyboard line framed by the RETURN has been displayed on the screen.
  12. Retrieve characters from the current logical screen line, working backwards from the end of the logical line until a character is found that is not a space. Each time the previous character is examined, DEY. When the nonspace character at the end of the logical screen line is found, the Y register equals the number of characters in the screen line (counting from 0).

## Screen Routines

---

13. INY to point to one position past the last nonspace character on the line, and store this in C8, the number of characters in the line, including the RETURN.
14. Set 0292, the screen scroll flag, to zero (scrolling enabled).
15. Set D3, cursor position within the logical line, to zero.
16. Set D4, quote mode flag, to zero, indicating that characters currently being processed are not within quotes.
17. If C9, the physical line number the cursor was on at entry to CHRIN, has its high bit on, BMI to E63A/E657 in the Get Character from Keyboard or Screen routine. If C9 = \$FF, the physical line has changed. Since we know the physical line has changed, the remaining steps are unnecessary. C9 could have its high bit on (could be \$FF) if C9 was 0 and the screen was scrolled. Or, C9 could have become 0 because of several cursor downs.
18. LDX D6, the current physical screen line cursor is on.
19. JSR E6ED/E719 to set (D1) to point to the start of the logical line the cursor is on. (Version 3 of 64 ROM only: JSR E591. At E591, JMP E6ED if the current physical screen line is not the same physical line as when the routine was entered. Thus, (D1) is reset only if the physical line changes.)
20. If the current physical screen line (after RETURN has been entered) is not the same physical line as when this routine was called, branch to E63A/E657.
21. If the current physical screen line (after RETURN has been entered) is the same physical line as when this routine was called, set D3 from CA .
22. Compare CA to C8, the current screen line length.
23. If CA, the current cursor column, is less than C8, branch to E63A/E657 to read the screen line. Thus, CA is used to determine whether any new characters were typed on this line and need to be reconverted from screen code to ASCII.
24. If CA, current cursor column, is greater than or equal to C8, branch to E65D/E691 to reset D0 to indicate input from the keyboard and to exit. For example, if changes are made to a line but RETURN is never entered on that line, it disregards the changes made in that line.

## **If Quote Key Detected Then Flip Quote Flag E684/E6B8-E690/E6C4**

### **Called by:**

JSR at E657/E68A in Get Character from Keyboard of Screen,  
JSR at E740/E76B in Main Screen Editor.

Compare the character passed in the accumulator to \$22, the code both for an ASCII quote key and for the screen quote character. If the character is a quote ("), flip the flag at D4 that indicates quote mode status. A resulting D4 value of 0 indicates normal mode and a value of 1 indicates quote mode.

### **Operation:**

1. Compare the accumulator to \$22, which is the code for an ASCII quote key and for the screen quote character.
2. If not equal, branch to step 5.
3. If equal, Exclusive OR D4 with \$01, thus flipping its value from 0 to 1 or from 1 to 0.
4. Restore accumulator to value of \$22.
5. RTS.

## **Set Color and Store Character on Screen EA13/EAA1-EA1B/EAA9**

### **Called by:**

JSR at E5E5/E5FF in Get Characters Until RETURN Key Detected, JSR at E6A3/E6D6 in Display Screen Code.

### **Operation:**

1. TAY to save the accumulator during this procedure.
2. Set CD to 2. The IRQ interrupt handler checks CD to determine when to blink the cursor.
3. JSR EA24/EAB2 to set the pointer to the color nybble for this screen byte.
4. TYA to restore the accumulator to its value at entry.
5. Fall through to EA1C/EAAA.

## **Display Byte in Accumulator on Screen EA1C/EAAA-EA23/EAB1**

### **Called by:**

JSR at EA5F/EAEF in IRQ Interrupt Handler; fall through from Set Color and Store Character on Screen.

Display the screen code from the accumulator at the current screen location of the cursor, and set the color of the corresponding nybble in color memory to the color of this byte.

### **Entry requirements:**

The accumulator should contain the screen code for the character to be displayed on screen. The X register should contain the value for the foreground color for this character. (D1) should point to the start of the logical screen line the cursor is on. D3 should hold the cursor position within the logical screen line. (F3) should point to the start of the line in color memory that corresponds to this logical screen line.

### **Operation:**

1. LDY from D3, the cursor position within this logical line.
2. STA (D1), Y. Put the screen code into screen memory and thereby display the character on the screen.
3. Store the X register value into (F3),Y. The color nybble location for this screen byte is thus set.
4. RTS.

## **Set Pointer to Color Nybble**

**EA24/EAB2-EA30/EABE**

### **Called by:**

JSRs at E75F/E78B and E807/E84E in Main Screen Editor, JSR at E9E0/EA6E in Set Color Memory Pointers for Moving Line, JSR at EA04/EA92 in Clear Screen Line Cursor Is On, JSR at EA18/EAA6 in Set Color and Store Character on Screen; JSR at EA4F/EADD in IRQ Interrupt Handler.

The pointer to the color nybble corresponding to the first character of the screen line the cursor is on, (F3), is set based on the pointer to the current screen line and the starting address of color memory.

Consider two examples of how this procedure works (using the VIC in these examples):

If (D1), pointer to current screen line, is 102C, then F3 is set to \$2C (the low byte of the pointer value), while the value for F4 is calculated as follows: \$10 (the high byte of the pointer value) is ANDed with \$03, then it is ORAed with \$94 for a result of \$94. So (F3) is \$942C.

If (D1) is 1E2C, then F3 is set to \$2C, while F4 is calculated as follows: \$1E is ANDed with \$03 (this leaves a bit value of 0000 0010), then ORAed with \$94 (a bit value of 1001 0100) for a result of \$96. Thus, (F3) is \$962C.

### Operation:

1. Get the low byte of the pointer to the line the cursor is on, D1.
2. Save in F3, the low byte of the pointer to color nybble corresponding to the start of current screen line.
3. Get the high byte of the pointer to the current screen line the cursor is on, D2.
4. AND \$03 to mask out all but the two lowest bits.
5. 64: ORA \$D8, the start of color memory (D800 is the first nybble of color memory). VIC: ORA \$94, the start of color memory (9400 is the first nybble of color memory).
6. STA in F4, the high byte of the pointer to the color nybble corresponding to the current screen line.

## Retrieve Character from Keyboard Queue E5B4/E5CF-E5C9/E5E4

### Called by:

JSR at E5E8/E602 in Get Characters Until RETURN Key Detected, JMP at F147/F1FE in GETIN Preparation.

The two routines that call this routine both check C6, the number of characters in the keyboard queue, before doing the call. If C6 is 0, indicating there are no characters in the keyboard buffer, this routine is not called. C6 is incremented by the Keyboard Scan routine each time the ASCII value for a key is stored in the keyboard queue. When this routine is called, the keyboard queue contains a character since C6 is nonzero.

Retrieve the ASCII value of the character at the start of the keyboard queue, at location 0277, and return this value in the accumulator.

Since the first entry in the keyboard queue has now been removed, move all the remaining values in the queue one byte toward the start of the queue. Also, decrement C6 to reflect the fact that one less value is now in the keyboard queue.

### Operation:

1. LDY with the first character in the keyboard queue, from location 0277.

## Screen Routines

---

2. Set X register to 0.
3. LDA with 0278,X.
4. STA 0277,X. Thus, move the character from position 0278 + X to the previous position in the keyboard buffer. In a move like this, where the move overlaps, it is important to move characters in the correct direction as is done here. If you instead initialized X to the current character count C6 and decremented X after each LDA-STA sequence, you would just propagate the last character in the buffer through all previous characters in the buffer.
5. INX, thus pointing the X register to the next higher address character in the buffer to be moved.
6. If the X register is now equal to C6, all characters have been moved; in this case fall through to step 7. If not equal, more characters remain to be moved, so branch back to step 3.
7. Decrement C6, the count of the number of characters in the keyboard buffer.
8. TYA. The accumulator now contains the ASCII character that was retrieved from the start of the keyboard buffer.
9. Enable IRQ interrupts.
10. CLC and RTS.

### **Main Screen Editor** **E716/E742-E87B/E8C2**

#### **Called by:**

JSR at E5CA/E5E5 in Get Characters Until RETURN Key Detected, JSR at E670/E6A3 in Get Character from Keyboard or Screen, JMP at F1D2/F282 in Determine Output Device.

This routine controls the flow of action for the screen editor, and it is also used by CHROUT. It determines whether the ASCII character passed to it in the accumulator is a valid screen code or a screen control character.

Screen control characters tested for are: INST, DEL, the cursor keys, the color keys, RVS ON, RVS OFF, CLR, HOME, Commodore/SHIFT keys together (switch case, lower to upper or upper to lower), CHR\$(8) (disable Commodore/SHIFT), and CHR\$(9) (enable Commodore/SHIFT). If one of these control keys is detected and the two conditions mentioned below are not in effect, this routine either performs the indicated function or calls another routine to perform the function.



Two other conditions also cause special action to be taken. If the character is a quotation mark (""), the current setting of the quote flag at D4 is reversed, and the quotation mark is displayed on the screen. Location D4 is also important, though, in determining whether to treat the normal control characters as control characters or to display them on the screen. If D4 is 1, indicating quote mode, only the DEL (delete) and RETURN keys are treated as valid control characters. Other control characters are displayed on the screen using the reversed version of the ASCII character for the indicated screen code.

The other special condition is when there are remaining inserts. Each time the INST key is entered, the count of outstanding inserts is incremented. This routine first checks for and processes a RETURN keypress. Otherwise, it calls the INST key handler. Thus, you can have more than one insert. Next, it checks to see if there are any outstanding inserts. If there are, it stores the ASCII value passed in the accumulator to the screen in the space opened up by the insert. Then it checks to see if an INST key was entered. Since the check for outstanding inserts is made before the check for the DEL key, the DEL key can be included in the screen line (if there are outstanding inserts). To display the DEL key (reverse video T), you must enter it before hitting RETURN for the line. This display of the DEL key as reverse T works either within or outside of quotes. However, if you leave the reverse T in a line without surrounding quotes, you get a syntax error for that line.

This routine does not display characters on the screen. Rather, it calls other routines to display them and to convert between screen and ASCII codes. The computer tests the ASCII values to determine what range the code is in and what conversion the ASCII code must go through to become a screen code. The other function of this routine, as mentioned above, is to take action for the control keys.

### **Operation:**

1. Push character to be displayed from the accumulator onto the stack, and save it in D7 as well.
2. Save the contents of the X and Y registers on the stack.
3. Set D0 to 0, indicating input from keyboard the next time the Get Character from Keyboard or Screen routine is executed.

## Screen Routines

4. Load Y register from D3, the cursor position within the logical line.
5. Load accumulator from the value that was stored into D7 in step 1.
6. If the value in the accumulator has its high bit off (ASCII key value < 128), branch to step 8.
7. If the value in the accumulator has its high bit on (ASCII key value >= 128), jump to step 43 to handle the ASCII values 128–255.
8. If the accumulator contains \$0D, the ASCII value for RETURN, JMP E891/E8D8 to handle the RETURN key.
9. If the accumulator value < \$20, branch to step 16. ASCII characters under \$20 are normally considered control characters.
10. If the accumulator value < \$60, branch to step 13 to handle ASCII characters from \$20–\$5F (32–95).
11. If the accumulator value >= \$60 and < \$80, AND \$DF, turning off bit 5. For example, for ASCII code 115 (heart), the following conversion would be performed:

115 decimal	0111 0011
AND #\$DF	<u>1101 1111</u>
	0101 0011

The result is \$53 (decimal 83), the screen code for the heart symbol.

12. BNE to step 14. This should be an unconditional branch since no keys in the Commodore keyboard decoding routines return the ASCII value of 0.
13. Branch here to handle ASCII characters from \$20–\$5F (32–95). AND \$3F, turning off the two high bits. For example, the ASCII code 57, which represents the character 9, is converted by:

57 decimal	0011 1001
AND \$3F	<u>0011 1111</u>
	0011 1001

This yields \$39 (decimal 57), the screen code for 9.

Another example for the character A, ASCII code 65, is as follows:

65 decimal	0100 0001
AND \$3F	<u>0011 1111</u>
	0000 0001

## Screen Routines

---

This produces \$01 (decimal 1), the screen code for A.

14. Branch here for ASCII keys  $\geq$  \$60 and  $<$  \$80; also fall through from step 13 for ASCII keys from \$20–\$5F. JSR E684/E6B8 to test for the quote character and flip the quote flag if the quote character is found.
15. JMP E693/E6C7 to display the converted ASCII character on the screen.
16. Branch here from step 9 to handle ASCII keys with values under \$20, normally ASCII control keys.
17. LDX D8, which contains the number of outstanding inserts. If there are no outstanding inserts, branch to step 19.
18. JMP E697/E6CB if outstanding inserts exist. If an outstanding insert is pending, ASCII characters with values under \$20 are displayed on the screen in the next outstanding insert position. To display an ASCII character, first ORA \$80 so that its reversed value is displayed. ASCII values under \$20 normally produce some action on the screen display or on the cursor. For example, the ASCII value for the WHT key is 5, or 0000 0101. ORA this value with \$80 to produce the screen code 1000 0101 (\$85), which is displayed on the screen as reverse E.

The ORA \$80 is actually done by the routine for ASCII characters 0–31. If there are outstanding inserts, the ASCII code is ORed with \$80 to display it as the reverse of the screen codes for 0–31.

Since the test for the RETURN key occurs before the test for outstanding inserts, the RETURN key is one control key that cannot be displayed on the screen in the outstanding insert area.

19. If accumulator doesn't hold a RETURN and no inserts are outstanding, test for other control key values.

The next value compared is \$14, the ASCII value for the DEL (delete) key.

If the current value is not a DEL keypress, branch to step 37. If a DEL key, continue with following action in steps 20–36.

20. TYA. The Y register contains the value of D3 (the column the cursor was on within the current logical screen line) on entry to the routine. If Y is nonzero (if the cursor is not on the first column of the logical screen line), branch to step 23. Remember that the leftmost column of the screen line is 0.

## Screen Routines

---

21. If the column the cursor was on was 0, JSR E701/E72D to see if the current physical screen line is 0. If it is 0, it is impossible to retreat to the previous physical screen line, so exit from the routine. If the physical screen line is not zero, the subroutine decrements the physical screen line length, resets the screen line link pointers and the length of the current logical screen line, and resets D3 to the length of the current logical line.
22. JMP to step 36 to insert a space character at the end of the current logical line.
23. Branch here if cursor column was not zero.  
    DEY and STY D3, thus decrementing the indicator of the cursor position within the logical screen line.  
    JSR EA24/EAB2 to set the address for the start of the line that the cursor is on in color memory, (F3).
24. INY to point to the next character in current logical screen line to be moved.
25. LDA (D1), Y to load this next character in the current logical screen line.
26. DEY, thus backing up to the previous character position.
27. STA (D1),Y, thus shifting the character just loaded one position to the left on screen.
28. INY to prepare for retrieving the color for the character just moved in step 27.
29. Load the color of the character just moved with LDA (F3),Y.
30. DEY to point to the character just moved one position to the left on the screen.
31. STA (F3),Y to restore the color for the moved character.
32. INY to point to the next character on the line to be overwritten (deleted). This INY points to the next position to be overwritten, while the INY in step 24 points to the character to move to this position.
33. CPY D5 to see if it is now pointing to the end of the logical screen line.
34. If not, it isn't finished moving all characters one position to the left, so branch back to step 24.
35. If equal, all characters from the place where the cursor was when the DEL key was entered to the end of the logical screen line have been shifted one position to the left.
36. Branch here from step 22, or fall through from step 35.  
    Store a space at the last position in the logical screen line,

## Screen Routines

---

using the current foreground color value, 0286. From here BPL E7CB/E7F7. This should be an unconditional branch, since the color value loaded into the accumulator should always be less than 128. At E7CB/E7F7, JMP E6A8/E6DC to exit from the routine without displaying any character on the screen.

37. Branch here from step 19 if the DEL key was not pressed. See if D4 = zero, indicating not in quote mode. If not in quote mode, branch to step 38.

If D4 is nonzero, indicating quote mode is active, JMP E697/E6CB. With quote mode on, the ASCII value in the accumulator at entry (now limited to one of the following keys: RVS ON, HOME, CRSR right, CRSR down, WHT, RED, GRN, BLU) is displayed as described above. See the explanation in step 18 for handling outstanding inserts to see an example of the conversion from control character to screen code.

38. If the character is not within quotes, test for a value of \$12, the RVS ON key. If not equal, branch to step 39. If equal, set C7 to \$12 to indicate characters are to be displayed on the screen with the high bit on in the screen code, thus producing a reversed character.
39. Test for a value of \$13, the HOME key. If not equal, branch to step 40. If equal, JSR E566/E581 to move cursor to top left of screen.
40. Test for a value of \$1D, the cursor right key. If not equal, branch to step 41.

If equal, INY. Since the Y register contained the column the cursor was on in this logical screen line, this pointer now points to the next position to the right in the logical screen line. JSR E8B3/E8FA to see if D3, which contained the column the cursor was on before this advance of the Y register, indicates the cursor is positioned at the end of a physical screen line (*not* the end of a logical screen line). If it is positioned at the end of a physical screen line, increment D6, the current physical screen line, unless the current line was already line 25/23, in which case just exit without changing D6.

After the JSR E8BB/E8FA, reset D3 to the value from the Y register, thus incrementing D3 to indicate that the cursor has moved one position to the right on the logical screen line.

## Screen Routines

Now DEY and compare Y to D5, the current logical line length. This D5 value was unchanged by E8BB/E8FA. If the Y register (location of cursor before cursor right) < D5, branch to E7AA/E7D6.

However, if the above comparison shows that the cursor was located at the end of a logical screen line, merely moving the cursor to the next physical screen line is not adequate. Rather, DEC D6 to indicate the physical screen line change taken above is nullified, and JSR E87C/E8C3 which resets D6. The action taken in E87C/E8C3 depends on whether the cursor is already located on physical screen line 25/23. If it is already on that line, scroll one logical screen line off the top of the screen, and scroll noncontinued physical screen lines in at the bottom of the screen. The number of physical screen lines scrolled in is equal to the number of physical lines in the logical screen line scrolled off the top. Finally, E87C/E8C3 resets D6 to equal the number of the next noncontinued logical screen line. Upon returning from the JSR to E87C/E8C3, reset D3 to zero, pointing the cursor to the first column of this newly obtained logical screen line.

E7AA/E7D6: Branch here if cursor right is detected and the cursor is not at the end of a logical screen line. Also fall through to E7AA/E7D6 after obtaining the next logical screen line if the cursor was positioned at the end of the logical screen line when the routine was entered. JMP E6A8/E6DC to exit from the screen editor routines without displaying any character, as the cursor right key is a nondisplayable character unless within quotes or within outstanding inserts.

41. Test for \$11, the cursor down key. If not equal, branch to step 42.

If a cursor down key is detected, add \$28/\$16 (40/22 decimal) to the current cursor location within the logical screen line. Store this value in the Y register.

INC D6, the physical screen line the cursor is on.

Now compare the value obtained by adding 40/22 to the current cursor location to D5 (the length of the current logical line). If the current cursor location within the logical line + 40/22 is less than or equal to the end of the logical line, there is no need to move to the next logical

screen line, so branch to E7A8/E7D4 to STY D3, the cursor position within the logical line, and then JMP E6A8/E6DC to exit from the screen editor routines.

If adding 40/22 to the current cursor location results in a value greater than the current logical line length, nullify the previous addition by subtracting \$28/\$16 (40/22 decimal) from the calculated value of the current cursor location and storing the result in D3, and nullify the change to D6, the physical screen line with DEC D6. Then, JSR E87C/E8C3 to reset D6 to the next noncontinued logical screen line, and to scroll the screen if necessary. Finally, JMP E6A8/E6DC to exit from the screen editor routines.

42. JSR E8CB/E912 to test for one of the color keys (BLK, WHT, etc.) and set 0286 to a foreground color if a valid color key code is found. The accumulator value is tested by comparing it to values in the table of valid color codes at E8DA/E921.

JMP EC44/ED21 to test for ASCII values for switching to the lowercase character set, switching to the uppercase character set, disabling character set switching, or enabling character set switching. If any of these values are found, perform the indicated action, then JMP E6A8/E6DC to exit from the screen editor routines without displaying any character on the screen.

43. Branch here from step 7 (or fall through from step 42) to handle ASCII characters from 128–255. (VIC only: NOP instructions appear from E800 to the next significant instruction at E815.)
44. Turn off the high bit in the accumulator.
45. Test for the value \$7F. If so, the original value was \$FF, the ASCII value for  $\pi$ . Substitute the other ASCII value for  $\pi$ , \$5E (\$DE with high bit on). Although there are two ASCII codes for  $\pi$ , only one is sent to the screen editor.
46. If the accumulator contains a value under \$20, branch to step 48 to handle the ASCII control characters from 128–159.
47. If the accumulator  $\geq$  \$20, a displayable character is in the accumulator. JMP E691/E6C5 to ORA \$40 and display the screen code on the screen. As an example, consider the ASCII code 161:

## Screen Routines

161	1010 0001
AND \$7F	<u>0111 1111</u>
	0010 0001
ORA \$40	<u>0100 0000</u>
	0110 0001

The result is \$61, 97 decimal, which is the screen code for the left half-block graphics character.

48. Test for a value of \$0D. (Remember in the following comparisons that the original high bit has been turned off.) If not equal to \$0D, branch to step 49. If equal to \$0D (SHIFT-RETURN), JMP E891/E8D8 to handle the RETURN key.
49. Test D4 to see if quote mode is on (if the character currently being processed is within quotes). If quote mode is on, branch to step 57 to ORA \$40 and display the character on the screen. If quote mode is off, fall through to step 50.
50. Test for a value of \$14, the INST key. If not equal, branch to step 56.
51. If the INST key was detected, LDY D5, the length of the logical line. Then LDA (D1),Y to retrieve the last character in this logical line. (D1) is the pointer to the start of the logical line the cursor is on.
52. See if this last character on the logical line is a space. If not, branch to step 53. If the last character is a space, see if the cursor is on the last character in the line. If it is, compare the current logical screen line length to \$4F/\$57 (79/87 decimal) to see if it is trying to insert at the very end of the maximum length of a logical line. If the cursor is not on column 79/87 in the logical screen line, JSR E965/E9EE to open up a blank screen line following this logical screen line to allow inserts. However, if the current length of the logical screen line is already 79/87, no further insertions can be made. Branch to step 55, which jumps to exit from the screen editor routines without displaying anything to the screen.
53. If the cursor was not on the end of the current logical screen line, branch here. Also, fall through to here if the cursor was on the end, but another physical line could be added to the logical screen line.

LDY with the current logical screen line length from D5.  
JSR EA24/EAB2 to set the pointer to the location in



color memory that corresponds to the start of the screen line in screen memory, (F3).

The following sequence shifts characters (from the end of the screen line back to the current location of the cursor) one position to the right on the screen:

DEY. Y now points to the character to be shifted one position to the right.

LDA (D1),Y. Retrieve this character.

INY. Y now points one position to the right of the character just loaded.

STA (D1),Y. Store the character in this position one location to the right of where it was taken.

Do the corresponding move for the color of this character, using (F3) to point to the location in color memory:

DEY. Y now points to the color of the character to be shifted one position to the right.

LDA (F3),Y. Retrieve the color.

INY. Y now points one position to the right of the color of the character just loaded.

STA (F3),Y. Store the color of the character in this position one location to the right of where it was taken.

DEY, so that the Y register now points to the position from which the character was just moved. Compare this location to the cursor location upon entry to this routine, stored in D3. If not equal, more characters need to be shifted to the right, so branch to the DEY at the beginning of this loop.

If Y register is equal to the cursor location, store the screen code for a space character at this location. Thus, if INST is entered, a space character is displayed on the screen. Color the space using the value for the current foreground color, 0286.

54. Increment D8, the number of outstanding inserts.

55. JMP E6A8/E6DC to exit from the screen editor without displaying anything on the screen.

Since quote mode is checked for before checking for an INST key, if quote mode is active, a space will not be inserted. Rather, the reverse of the screen code equivalent to \$94 (the ASCII code for INST) is displayed. When in quote mode, \$94 AND \$7F, which is \$14, OR'ed with \$40, yields a screen code of \$54 (decimal 84), which is a thin vertical-line graphics character.

If quote mode is not active, multiple inserts can be made. Each time an INST is detected, another space is inserted at the cursor location. The limit on the number of insertions is the difference between the current length of the logical line and its maximum length (80/88 characters).

After an insertion is performed, the cursor does not move one space to the right, but rather stays where it was when INST was typed.

56. Test D8 to see if there are any outstanding insertions to be made. If not, branch to step 58. If outstanding inserts remain, fall through to step 57.

57. Branch here from quote mode or fall through if outstanding inserts remain. ORA \$40 to convert an unprintable control code to a displayable screen code.

JMP E697/E6CB, which does ORA \$80 to display the code as a reverse video character.

58. Test for a value of \$11, the cursor up key. If not, branch to step 63. If it was, LDX D6, the current physical screen line the cursor is on. If the cursor is on line 0, it is impossible to move the cursor up any higher on the screen, so exit the screen editor.

59. If the cursor is not on line 0, DEC D6, the current physical screen line.

60. LDA D3, the column the cursor is on in this logical screen line, and subtract \$28/\$16 (40/22 decimal). If carry is clear after this subtraction, a borrow was required, meaning that the cursor was on the first physical line in a logical screen line. Branch to step 62 if carry is clear.

61. If the carry was set, a borrow was not required, meaning that the cursor was originally on physical screen line 2 (64) or 2, 3, or 4 (VIC) of the logical screen line. After subtracting 40/22, the cursor now points to the same column in the previous physical screen line—1 (64) or 1, 2, or 3 (VIC). BPL to E871/E8B8, which jumps to E6A8/E6DC to exit. This should be an unconditional branch, as the accumulator should not be greater than 39/65, the length of 1/3 physical screen line(s), and the high bit is thus 0.

62. If the cursor up moves the cursor out of physical line 1 of a logical screen line, JSR E56C/E587 which resets the screen line link table, establishes the pointer to the start of

the previous logical screen line, and adjusts the cursor column value and the length of the current screen line.

E56C/E587 checks for noncontinued screen lines, starting with the current location of the physical screen line pointed to by D6 and working backward until it finds an uncontinued screen line. Each time it moves up a physical screen line, 40/22 is added to the column the cursor is on. Here's an example for the VIC: If E587 is called with the cursor pointing to column 5 of physical line 10, and if the previous noncontinued line is line 8 (the start of the previous logical line), add 22 to 5 to reset the cursor pointer to 27, pointing correctly to the sixth column (remember offset from 0) of physical line 9, which is the second physical screen line of the previous logical line.

What if none of the previous physical screen lines have their high bit on, indicating a noncontinued line (the start of a logical line). This condition should not occur, but conceivably it could happen. If this condition did exist, E56C/E587 would keep looking for a logical screen line until it reached physical screen line 0, at which point it would exit the search for a logical screen line. While doing this search, though, it would have added 40/22 to the current cursor position D3 each time it moved up another physical screen line. You could use a test for D3 being greater than 79/87 to make certain that the screen editor routines for cursor up have not gone awry. The Kernal does not make such a check.

After the JSR E56C/E587, exit the screen editor routines.

63. Test for a value of \$12, the RVS OFF key. If it is found, reset C7 to 0, indicating that screen codes are to be displayed in their normal, nonreversed mode. Whether or not RVS OFF is found, continue with step 64.
64. Test for a value of \$1D, indicating a cursor left key. If not equal, branch to \$E8B1, step 67. If a cursor left key is detected, see if the current cursor column is column 0 of this logical screen line. If so, branch to step 66. If not, continue with step 65.
65. JSR E8A1/E8E8 to see if the column the cursor is on is the first column (40/22, 44, or 66) of a physical screen line (line 2/line 2, 3, or 4) within this logical screen line. If so, DEC D6, the current physical screen line.

DEY and store in D3, thereby decrementing the column the cursor is on.

JMP E6A8/E6DC to exit from the screen editor routines without displaying anything on the screen.

66. Branch here if the cursor was on column 0 of the logical screen line. JSR E701/E72D, which first examines whether the cursor is on physical screen line 0 and exits if it is. If the cursor is not on physical screen line 0, decrement the physical screen line, D6, and JSR E56C/E587 to reset the screen line link table and the current logical line length, and to adjust D3 to the current logical line length. D3, the current cursor location, now points to the last character in the previous screen line. JMP E6A8/E6DC to exit.
67. Test for a value of \$13, the CLR key. If the keypress is not CLR, branch to step 68. If it is a CLR, JSR E544/E55F to clear the screen and reset the screen line link pointers. JMP E6A8/E6DC to exit.
68. If the key value in the accumulator was not one of the control codes, ORA \$80, which restores the accumulator to its original value before the AND \$7F in step 44.
69. JSR E8CB/E912 to test whether the key value is one used for changing the foreground color.
70. JMP EC4F/ED30 to see if the character indicates switching to the uppercase character set, disabling character set switching, or enabling character set switching. If so, perform the requested action, if currently allowed by the editor. Exit by jumping to E6A8/E6DC.

### Display Screen Codes

#### E691/E6C5-E6A7/E6DB

##### Called by:

JMP at E7E0/E827 in Main Screen Editor; alternate entry at E693/E6C7 by JMP at E743/E76E in Main Screen Editor; alternate entry at E697/E6CB by JMP at E74A/E775, E783/E7AE, and E830/E876 in Main Screen Editor.

This routine is called by the Main Screen Editor routine after an ASCII key value has been detected. Any conversion of ASCII codes to screen codes may be handled either by the Main Screen Editor or by entering this routine at the appropriate point.

This routine tests for the reverse mode flag. If it is on, the character passed to it (entry points E691/E6C5 or E697/E6CB) is displayed as the reverse screen character. If called with inserts outstanding, the count of outstanding inserts is decremented.

Finally, the routine calls other routines to display the screen code and to advance the cursor to the next position in this logical line, or to advance the cursor to the next logical line if necessary.

### **Entry requirements:**

The accumulator should hold the screen code, or a partially converted ASCII code, for the character to be displayed on the screen. D8 holds the number of outstanding inserts. C7 should be 0 if characters are to be displayed without reversing them; it should contain a nonzero value (18 is typical) if codes are to be displayed reversed. 0286 should hold the current foreground color.

### **Operation:**

1. E691/E6C5: ORA \$40. See comments in the description of the Main Screen Editor routine for the reason this is necessary.
2. E693/E6C7: See if the reverse flag, C7, is on (nonzero). If not, branch to step 4. If so, fall through to step 3.
3. E697/E6CB: ORA \$80 to set the high bit of the screen code to 1 and display the screen code character in reverse video. All screen codes >128 appear as reverse video.
4. If D8 is zero, indicating there are no outstanding inserts, branch to step 6.
5. DEC D8 if outstanding inserts remain, since the character is being displayed in insert mode.
6. LDX 0286 the current foreground color. This color will be used in displaying the screen code.
7. JSR EA13/EAA1 to set the color memory pointer to the screen code and display the screen code on the screen with this color.
8. JSR E686/E6EA to advance the cursor on the screen.
9. Fall through to E6A8/E6DC, Exit from Screen Editor Routines.

### **Exit from Screen Editor Routines**

**E6A8/E6DC-E6B5/E6E9**

#### **Called by:**

Main Screen Editor routine to exit after any of these ASCII keys are detected: INST, CLR, cursor right, down, or left; also called for all exits from the Test for Character Set Switch routine.

This exit routine is called by several routines in the screen editor that handle control keys or in situations where no character is to be displayed on the screen. The routine pops off the registers that were saved on the stack at entry to the Main Screen Editor routine. To maintain a correct stack, this routine must be called from screen editor routines even if no character is to be displayed. The routine that displays characters to the screen also falls through to this exit routine after displaying the character and advancing the cursor.

This routine also checks to see if any outstanding inserts remain, and if so, it makes certain that the quote mode is turned off. So, insert mode has priority over quote mode. However, during the Main Screen Editor routine, the quote mode is checked before the insert mode. Thus, if you are in quote mode, you can't insert spaces, and if you have outstanding inserts, you can't be in quote mode.

Finally, the carry is cleared and interrupts enabled before exit.

#### **Operation:**

1. Restore Y register from stack.
2. Test D8 for outstanding inserts, and if there are, reset D4 to 0 to turn off quote mode if it was on.
3. Restore X register and accumulator from stack.
4. CLC, enable interrupts, and RTS.

### **Advance Cursor and Scroll or Insert Blank Lines**

**E6B6/E6EA-E700/E72C (VIC: also ED5B-ED68)**

#### **Called by:**

JSR at E6A5/E6D9 in Display Screen Codes; alternate entry at E6DA/E70E by JMPs at E97F/EA05 and E9C3/EA46 in Insert Blank Line.

Advance the pointer to the cursor location within the logical screen line. If the cursor has not reached the end of the logical line, exit.

However, if advancing the cursor moves it to the end of the logical screen line, try to move the cursor to column 0 of the next physical screen line.

If the current logical line has not already reached its maximum length, make this next physical screen line a continuation line for the current logical line. Certain special conditions are checked for in getting the next physical screen line. For example, if location 0292 contains a 0, the editor allows the last physical screen line to be removed from the screen if another physical screen line is to be added to the current logical line.

If the current logical screen line has reached its maximum length, reset the pointer to the next physical screen line and reset the logical line length to 0. If the next physical screen line is 25/23, scroll the screen.

### Operation:

1. JSR E8B3/E8FA to see if the current location of the cursor is on the end of physical line 1/line 1, 2, or 3 of a logical line. If the cursor is on the final column of a physical screen line, as long as the physical screen line is not on line 25/23, increment the physical screen line location, D6.
2. Increment the column the cursor is on within this logical line, D3.
3. If the current logical line length, D5, is greater than or equal to this new cursor location within the logical line, D3, branch to step 23 to RTS.
4. If, however, the column the cursor is on is greater than the current length of the logical screen line, continue with step 5.
5. If the current logical screen line length is 79/87, branch to step 19; otherwise, fall through to step 6.
6. LDA 0292.
7. If 0292 is 0, indicating that screen scrolling is allowed, branch to step 9; otherwise, continue with step 8.
8. JMP E967/E9F0. The routine at E967/E9F0 will either find a noncontinued line between the current physical line and the bottom of the screen and reset the line to be a continuation line, or if it reaches line 24/22 without finding a noncontinued line, it will scroll the screen. RTS from the JMP to exit this routine.
9. Test D6 to see if the current physical screen line is now line 25/23. If < 25/23, branch to step 13.

## Screen Routines

---

10. JSR E8EA/E975 to scroll a logical line off the top of the screen and to scroll a logical line in on the bottom of the screen.
11. Decrement the current physical screen line, D6.
12. LDX D6. The X register will be used as an index into the screen line link table in step 13.
13. E6DA/E70E: Using the X register as an index and D9 as the base of the screen line link table, shift the table entry addressed by D9,X left one bit to shift out the high bit, and then shift it right one bit, shifting a 0 back into the high bit. These two shifts make the table entry indicate that the corresponding screen line is a continuation line, since the byte now has its high bit off.
14. VIC only: JMP ED5B, which is a patch area. At ED5B, continue with step 15.
15. Now increment the X register, retrieve the next byte in the screen line link table pointed to by D9,X, turn on its high bit with ORA \$80, making this next physical screen line a noncontinued line. DEX so that now the X register points back to the continued screen line. LDA D5, the current screen line logical length, and CLC.  
VIC only: JMP E715 to return to the main routine at step 16.
16. Increase D5, the current logical screen line, by \$28/\$16 (40/22 decimal), as another physical screen line has been added to this logical screen line.
17. Now determine which physical screen line is the start of this logical screen line by working back up through the screen line link table from the current location pointed to by the X register. Once a noncontinued line is found, that physical screen line is also considered the start of the logical screen line.
18. JMP E9F0/EA7E to reset the pointer to the current logical screen line, (D1), using the X register value just obtained. RTS from the JMP to exit the routine.
19. Branch here from step 5. Decrement D6, the pointer to the start of the current physical screen line.
20. JSR E87C/E8C3 to see if the next physical screen line is line 25/23. If it is, scroll the screen. Set D6, the current physical screen line, from the value in the screen line link table indicating the next noncontinued line, and then reset



the screen line link table and the current logical screen line length.

21. Reset D3, the pointer to the cursor location within the logical screen line, to 0.
22. RTS.

### **Move Cursor to Previous Screen Line** **E701/E72D-E715/E741**

#### **Called by:**

JSRs at E754/E77F and E865/E8AB in Main Screen Editor.

Whenever a DEL or cursor left key is entered when the cursor is positioned on column 0 of a logical screen line, this routine is called.

The routine first tests to see if the current physical screen line is line 0. If so, it exits, since it is impossible to move the cursor to a previous line.

If the current physical screen line is not 0, the current physical screen line is decremented. Next, JSR E56C/E587 to reset the screen line link pointers, reset the pointer to the start of the logical screen line (D1) to point to the start of the previous noncontinued line, and reset the current logical line length D5. Upon return from the JSR, reset D3, the cursor location within the logical screen line, to be equal to D5, thus pointing to the end of the previous logical screen line.

#### **Operation:**

1. If the current physical screen line, D6, is not 0, branch to step 3.
2. If the current physical screen line, D6, is 0, set D3 to 0, pull the top two bytes off the stack to remove the return address of this routine, and then branch to E6A8/E6DC to exit the screen editor routines without displaying anything on the screen.
3. Decrement D6, so that it points to the previous physical screen line.
4. JSR E56C/E587 to reset the screen line link table, the pointer to the start of the logical screen line (D1), and the logical screen line length, D5.
5. Upon returning from the JSR in step 4, store D5 into D3, thus setting the cursor position within the logical screen line to the end of the logical screen line.

### Advance Cursor to Next Screen Line

E87C/E8C3-E890/E8D7

#### Called by:

JSR at E6FA/E725 in Advance Cursor and Scroll or Insert Blank Lines, JSRs at E7A4/E7CF and E7C9/E7F4 in Main Screen Editor, JSR at E89C/E8E2 in Handle RETURN Key.

This routine is called when any of the following conditions occur: the cursor moves to the next screen line because the cursor has moved past the end of the logical screen line (in the routine which advances the cursor on the screen after displaying a character); the cursor moves to the next screen line because the cursor right key has moved the cursor past the end of the logical screen line; the cursor down key is entered, moving the cursor down one line; the RETURN key (or SHIFT-RETURN) is entered.

D6 is changed to point to the next logical screen line (the next physical screen line that is noncontinued, which has a link with its high bit on). If a noncontinued screen line is not found upon reaching line 25/23, the scroll routine is called to scroll the screen. Exit this routine by jumping to reset the screen line link table, the length of the current logical line, and the pointer to the start of the current logical line.

This routine also does a LSR C9, the location that holds the logical line number on entry to CHRIN. Each time there is a cursor down, RETURN, SHIFT-RETURN, cursor right past the end of the logical line, or display on the screen past the end of the logical line, this LSR takes place. Consider the case where the cursor is positioned on physical screen line 9, which is the first of two lines that make up one logical line. C9 contains 8 when it is positioned on the ninth physical line (offset from 0) after pressing RETURN on the previous line. After another RETURN, if you examine C9 it will hold a value of 10 (pointing to the eleventh physical screen line). However, by examining location C9 during the RETURN key process, it appears that in this example C9 is reset to 4 before becoming 10. The sequence of change does make sense, since there is an LSR C9, and 8 decimal (binary 0000 1000) shifted right yields 4 decimal (binary 0000 0100). The next call of CHRIN then resets C9 to the current physical screen line, the value of which is stored in D6. C9 is also decremented when the screen is scrolled—thus possibly setting C9 to \$FF after successive calls

of the routine have done enough LSRs to make  $C9 = 0$  (after several cursor downs, for example).  $C9$  is tested for having its high bit on, which could be the case if the screen has scrolled.

### Operation:

1. LSR  $C9$ . (See the discussion above.)
2. LDX from  $D6$ , the current physical screen line.
3. INX, thus preparing to move the cursor to the next physical screen line.
4. If  $X$  is not equal to  $25/23$ , branch to step 6.
5. If  $X$  is equal to  $25/23$ , JSR  $E8EA/E975$  to scroll the screen.
6. Retrieve the byte from the screen line link table that starts at  $D9$ , indexed by  $X$  register.
7. If this screen link byte has its high bit off, this next line is a continued line, in which case branch to step 3 as this routine is attempting to advance the cursor to the next logical line, not the next physical line.
8. If the screen link byte has its high bit on, indicating the start of the next logical screen line, STX  $D6$ , resetting the current physical screen line to the next logical screen line.
9. JMP  $E56C/E587$  to reset the screen line link table entries, the pointer ( $D1$ ) to the start of this logical screen line, and the current logical line length  $D5$ .

### Handle RETURN Key $E891/E8D8-E8A0/E8E7$

#### Called by:

JMPs at  $E72E/E75A$  and  $E7E7/E82$  in Main Screen Editor.

Whenever the Main Screen Editor routine detects the RETURN key or the SHIFTed RETURN key, this routine is called to perform the following actions: set the number of outstanding inserts to 0, turn reverse mode off, turn quote mode off, and set the current cursor column to 0. The routine to advance the cursor to the next logical screen line is then called. Finally, jump to the Exit from Screen Editor routine.

#### Operation:

1. Store 0 into the following locations:  $D8$ , the number of outstanding inserts;  $C7$ , flag to indicate whether characters are displayed in normal or reverse mode;  $D4$ , flag to indicate whether or not characters are within quotes;  $D3$ , the cursor position within the logical screen line.

2. JSR E87C/E8C3 to reset the pointer to current physical screen line to point to the next logical screen line.
3. JMP E6A8/E6DC to exit from the screen editor routines.

### **Decrement Screen Line Pointer If Cursor Moves Left to New Line**

**E8A1/E8E8-E8B2/E8F9**

#### **Called by:**

JSRs at E759/E785 and E85B/E8A2 in Main Screen Editor.

This routine is called from the Main Screen Editor routine when a DEL or a cursor left key is detected and the cursor is not currently on column 0 of the logical line. This path then tests to see if the cursor position indicated by D3 is 22, 44, or 66 (VIC) or 40 (64). If it is, the cursor is on the first column of physical line 2, 3, or 4 (VIC) or 2 (64) of the current logical screen line. Therefore, decrement D6, the pointer to the physical screen line, to indicate that the impending leftward movement of the cursor will carry it up to the previous physical line. On return to the Main Screen Editor routine, the cursor location within the logical screen line will be reset.

If the cursor was on column 0, the Main Screen Edit routine instead calls the routine to move the cursor to the end of the previous logical screen line.

#### **Operation:**

1. LDX \$02/\$04. Index of number of comparisons to be made.
2. LDA \$00. Entry value to start comparison with. However, since Main Screen Edit checks for  $D3 = 0$  before calling this routine, no match for 0 should ever occur here.
3. CMP D3, thus comparing the cursor location within this logical screen line to 0, 22, 44, and 66 (VIC) or 0 and 40 (64).
4. If current value of accumulator is equal to D3, branch to step 9.
5. Add \$28/\$16 (40/22 decimal) to the accumulator in preparation for the next comparison.
6. DEX.
7. If X register is not yet 0, branch to step 3. Example for VIC: The X register now contains 3 if the comparison with 0 was just made, 2 if the comparison with 22 was just made, 1 if the comparison with 44 was just made, or 0 if the comparison with 66 was just made.

8. If X register is 0, RTS as the cursor is not at the start of physical screen line 2, 3, or 4 (VIC) or 2 (64) of this logical screen line.
9. If the cursor is at the start of a physical line within the logical line, decrement D6, the current physical screen line the cursor is on, then RTS.

### **Increment Screen Line Pointer If Cursor Moves Right to New Line**

**E8B3/E8FA-E8C1/E911**

#### **Called by:**

JSR at E6B6/E6EA in Advance Cursor and Scroll or Insert Blank Lines, JSR at E798/E7C3 in Main Screen Editor.

This routine tests to see if the cursor position indicated by D3 is 21, 43, 65, or 87 (VIC) or 39 or 79 (64). If it is, the impending rightward movement will carry the cursor onto the next physical screen line. Therefore, increment D6, the current physical line the cursor is on, unless D6 contains 25/23, indicating that the cursor is already on the bottom line of the screen. On return to the Main Screen Editor routine, the cursor location within the logical screen line will be reset.

#### **Operation:**

1. LDX \$02/\$04. Index of the number of comparisons to be made.
2. LDA \$27/\$15. Entry value to start the comparison with is 39/21, the end of the first physical screen line within a logical screen line.
3. CMP D3, thus comparing the cursor location within this logical screen line with 39 or 79/21, 43, 65, or 87.
4. If current value of the accumulator is equal to D3, branch to step 9.
5. Add \$28/\$16 (40/22 decimal) to the accumulator in preparation for next comparison.
6. DEX.
7. If the X register is not yet 0, branch to step 3. For the VIC: The X register will contain 3 if the comparison with 21 was just performed, 2 if the comparison with 43 was just done, 1 if the comparison with 65 was just made, or 0 if the comparison with 87 was just completed. For the 64: The X register is 1 after the comparison with 39, or 0 after the comparison with 79.

8. If the X register is 0, RTS as the cursor is not on the end of physical screen line 1 or 2/screen line 1, 2, 3, or 4 of the current logical screen line.
9. Branch here from step 4 if the cursor is at the end of one of the physical lines within the current logical line. If D6 is 25/23, the cursor is already on the last physical line of screen, and thus it can't move down a line, so branch to step 11 to RTS.
10. Increment D6, the current physical screen line the cursor is on.
11. RTS.

### **Test for Color Key E8CB/E912-E8D9/E920**

#### **Called by:**

JSRs at E7CE/E7FA and E876/E8BD in Main Screen Editor.

Test to see if the current key value is BLK, WHT, RED, CYN, PUR, GRN, BLU, or YEL. For the 64, also check if the value is the Commodore key plus a color key, for the additional colors orange, brown, light red, dark grey, medium grey, light green, light blue, and light grey. If the current key is a color key, set 0286, the current foreground color, to the proper value for that color.

#### **Operation:**

1. LDX \$0F/\$07 to prepare for 16 possible comparisons on the 64 or 8 comparisons on the VIC. X is used to index the color code key table; when the match is found, the X register value is the color code for that key.
2. Compare the entry from the color code table at E8DA/E921, indexed by X, with the value in the accumulator.
3. If equal, branch to step 7.
4. If not equal, DEX.
5. If the X register is not 0, branch back to step 2, thus comparing the entries in the table from the end of the table to the start.
6. If the X register is 0, RTS, as no color key was detected.
7. If a match has been found, STX 0286. Location 0286 now contains the color value indicated by the color key.

### Scroll Screen

**E8EA/E975-E964/E9ED**

#### Called by:

JSR at E6D3/E707 in Advance Cursor and Scroll or Insert Blank Lines, JSR at E885/E8CC in Advance Cursor to Next Screen Line, JSR at E975/E9FD in Insert Blank Line.

This scroll routine moves one or more physical lines off the top of the screen until an entire logical line has been moved off the top of the screen, while moving the remaining physical lines all up one line.

The routine also resets the screen line link table to correctly reflect which lines are continued on the screen after the scroll.

The last line on the screen, line 25/23, is made a noncontinued line and is cleared. It is possible that more than one physical line will be scrolled in at the bottom of the screen; the number scrolled in is equal to the number of physical lines scrolled off the top.

The routine also zeros the byte that indicates the number of characters in the keyboard queue, effectively emptying it, and exits with the X register containing the current physical screen line, D6.

#### Operation:

1. Push AC, AD, AE, and AF onto the stack to preserve the values from these locations.
2. Set the X register to \$FF in preparation for step 4.
3. Decrement D6 (the current physical screen line), C9 (the saved physical line number at entry to CHRIN), and 02A5/F2 (the screen line link table temporary save value). Each time a physical screen line is scrolled off the top of the screen, these values are decremented.
4. INX. X will contain 0 after the first time through, and 1-24/1-22 on following passes.
5. JSR E9F0/EA7E to set the pointer to the start of the logical screen line, (D1), based on the current value of the X register.
6. If the X register value is greater than or equal to 24/22, branch to step 11, as the first 24/22 lines have been moved up one physical line on the screen, with the initial top physical line removed.

## Screen Routines

---

7. LDA ECF1/EDFE, indexed by X, retrieving the low byte of the address of this screen line, and then save it in AC.  
Examples: On the VIC, for X = 0 this retrieves \$16, and for X = 1 it returns \$2C ; on the 64, for X = 0 this yields \$28, and for X = 1 it returns \$50. Thus, the value retrieved is actually the low byte of the starting address of the next physical screen line.
8. LDA DA, X thus retrieving the high byte of the corresponding address from the screen line link table.
9. JSR E9C8/EA56 to convert the value in the accumulator to the high byte of the address of the screen line to be moved, set the pointer to the color memory start of this line, (F3), and then move all 40/22 columns of this physical screen line to the previous physical screen line.
10. BMI to step 4. This is an unconditional branch as the routine at E9C8/EA56 only exits when the BMI condition is true.
11. JSR E9FF/EA8D to clear the physical screen line the cursor is on, thus clearing physical screen line 25/23.
12. LDX \$00, preparing to index the screen lines from the top to the bottom.
13. LDA D9,X to retrieve the next entry from the screen line link table. AND \$7F to turn off the high bit of the entry.
14. LDY DA,X to retrieve the entry following the one loaded in step 13.
15. If the entry for the following line (from step 14) has its high bit off (if the physical line is a continuation line), branch to step 17.
16. If the entry has its high bit on to indicate a noncontinued line, ORA \$80 to set the high bit of the accumulator value as well.
17. STA D9,X. Update the first line link entry, making it either continued line or noncontinued line based on value in the following link table entry.

The following diagram illustrates the action this portion of the routine performs in resetting the link table values to indicate continued or noncontinued line for one particular screen map (for the VIC):



## Screen Routines

### Screen Scrolling (VIC example)

Physical Line	Original Link Values	After Move
0	continued	noncontinued
1	noncontinued	noncontinued
2	noncontinued	continued
3	continued	continued
4	continued	continued
5	continued	noncontinued
6	noncontinued	continued
7	continued	noncontinued
8	noncontinued	continued
9	continued	noncontinued
10	noncontinued	noncontinued
11	noncontinued	continued
12	continued	continued
13	continued	continued
14	continued	noncontinued
15	noncontinued	continued
16	continued	noncontinued
17	noncontinued	continued
18	continued	noncontinued
19	noncontinued	noncontinued
20	noncontinued	continued
21	continued	noncontinued
22	noncontinued	noncontinued

18. INX to prepare to update link byte for next line.
19. If X is not equal to \$18/\$16 (24/22 decimal, for line 25/23, since offsets are from 0), branch back to step 13 to adjust the next link table entry; otherwise, fall through to step 20.
20. Set line 25/23 to a noncontinued value.
21. Test the first line of the screen now to see if it is noncontinued. If it is a continuation line, branch back to step 2 to again scroll a physical line off the top of the screen. Thus, this scroll routine scrolls physical lines off the top of the screen until an entire logical line has been scrolled off the top.
22. If the first line is noncontinued, an entire logical line has been scrolled off the top; increment D6, the pointer to the current physical screen line, and also increment 02A5/F2, the temporary screen link byte. Since C9 is not incremented, a test for it being not equal to D6 can be used as an indication that the screen has been scrolled, although

these two values being unequal is also an indication that the cursor has moved to a different physical screen line than it was on at entry to CHRIN.

23. Reset the keyboard scan column to \$7F/\$FB to scan for the CTRL key.
24. If the row value from the keyboard scan is \$FB, indicating that the CTRL key was pressed, execute a delay loop; otherwise, reset the value for the keyboard scan to \$7F/\$F7 (the default value for detecting the STOP key) and branch to step 26. This added delay loop if the CTRL key is pressed makes it possible to use the CTRL key to slow screen scrolling during listings, etc.
25. Reset the number of characters in the keyboard buffer, C6, to 0.
26. Load X register with the current physical screen line number, D6.
27. Restore the original contents of locations AF, AE, AD, and AC from the stack and RTS.

### **Insert Blank Line**

**E965/E9EE-E9C7/EA55**

#### **Called by:**

JSR at E802/E849 in Main Screen Editor; alternate entry at E967/E9F0 by JMP at E6CA/E6FE in Advance Cursor and Scroll or Insert Blank Lines.

This routine is called to insert a blank line on the screen at the first noncontinued line following the cursor location, and then to make the blank line a continued line. The following lines on the screen are moved down one line.

If the next noncontinued line on the screen is not reached before line 25/23, the screen is scrolled and the bottom logical line inserted is the blank continued line.

#### **Operation:**

1. LDX D6, the current physical screen line the cursor is on.
2. E967/E9F0: INX. If entering at E967/E9F0, the X register holds either 0 if cursor is not on last column of a logical line, or D6-1 if the cursor is on the last column of the logical line.
3. Using the X register as an index into the screen line link table, search downward in the table for the first line that is

## Screen Routines

- noncontinued (the first line for which the high bit of the table entry is set to 1). Once one is found, fall through to step 4.
4. Save the noncontinued line number from the X register in 02A5/F2 as the first noncontinued line past the current location of the cursor.
  5. If this line number is less than or equal to 24/22, branch to step 10.
  6. If not, we have reached line 25/23, so JSR E8EA/E975 to scroll the screen.
  7. LDX 02A5/F2, thus reloading the X register with the value of the first noncontinued line, which is 25/23 in this case.
  8. DEX and DEC D6 to point to the previous physical screen line (24/22).
  9. JMP E6DA/E70E to make line 25/23 a continued line, and RTS to exit from the routine.
  10. If the first noncontinued line past the cursor location is less than or equal to 24/22, continue here by first pushing the contents of locations AC, AD, AE, and AF onto the stack to preserve their current values.
  11. LDX \$19/\$17 (25/23 decimal), to prepare to index the screen from the bottom up.
  12. DEX. Start with X = 24/22.
  13. JSR E9F0/EA7E to set (D1) to point to the start of this screen line.
  14. If X is less than or equal to 02A5/F2, the saved value of the first noncontinued line, branch to step 18.
  15. If X is greater than 02A5/F2, set AC, low byte of the screen line starting address, from the screen line link table at ECF0/EDFD. However, AC is set by LDA ECEF,X/EDFC,X. Thus, the value stored in AC is the low byte of the address of the previous screen line. Similarly, retrieve the high byte of the address from the screen line link table, starting at D8 rather than D9.
  16. JSR E9C8/EA56 to convert the value in the accumulator into the proper high byte of the address in AD and move the screen line pointed to by (AC) to the screen line pointed to by (D1), thus moving the screen line physically down one position on the screen. This direction of movement is why the index of the screen for this move must work from the bottom of the screen upwards.

## Screen Routines

17. Branch to step 12 to handle the next screen line until all screen lines up to 02A5/F2 (but not including the line pointed to by 02A5/F2) have been moved down one physical line.
18. JSR E9FF/EA8D to clear the screen line pointed to by 02A5/F2. Thus, a blank line has now been inserted into the screen.
19. Now change the screen line link table entries for the screen lines past 02A5/F2 in a manner similar to that used for resetting the screen line link table in steps 13–19 of the Screen Scroll routine. The difference here is that the accumulator is loaded with DA,X and the Y register with D9,X and updated value is DA,X. Thus, work upwards from the bottom of the screen in resetting the link table.
20. LDX 02A5/F2, thus the X register now points to the noncontinued blank line that has just been inserted.
21. JSR E6DA/E70E to make the line a continuation of the current logical line.
22. VIC: Restore the original values of locations AF, AE, AD, and AC from the stack and RTS.  
64: JMP E958 to restore AF–AC from the stack and RTS.

### Move Screen Line

**E9C8/EA56–E9DF/EA6D**

#### Called by:

JSR at E90E/E998 in Scroll Screen, JSR at E9A1/EA27 in Insert Blank Line.

The screen line pointed to by (AC) is moved to the screen line pointed to by (D1), along with the corresponding colors from the original line. At entry to this routine, the accumulator contains a value from the screen line link table. This value is converted to the high byte of a screen memory address by ANDing with \$03 and ORAing with the screen memory page.

Although the Kernal does not use any alternate entry points, if you want to use this routine yourself for moving screen lines, you can set (AC) to point to the line to be moved, (D1) to point to the destination, and then JSR E9CF/EA5D. If you want to save an image of the screen memory in another area of memory, you can set (AC) to point to the start of the screen, (D1) to point to the start of your save area, initialize

the X register to 0, and then call this alternate entry point. After each execution of the routine, add \$28/\$16 (40/22 decimal) to (AC) and (D1), and repeat this loop until the X register = 25/23; all 25/23 lines of the original screen will be saved in your temporary save area.

### **Operation:**

1. The accumulator contains the value of the high byte of the screen line link table (for example, \$9E).
2. AND \$03 to drop all but the two low bits. For example, \$9E AND \$03 = \$02.
3. ORA 0288, the high byte of the address of the start of screen memory. For example, \$02 ORA \$10 = \$12.
4. STA AD, thus setting the high byte of (AC), the pointer to the address of the line to be moved.
5. JSR E9E0/EA6E to set (AE), the pointer to the color memory address corresponding to the start of the source screen line, and (F3), the pointer to the color memory location corresponding to the destination screen line.
6. Using the X register as an index, and starting with an initial value of \$27/\$15 (39/21 decimal), move all 40/22 bytes from the original screen line to the destination screen line. Also move the corresponding color memory nybbles from origin to destination.

### **Set Color Memory Pointers for Moving Line E9E0/EA6E-E9EF/EA7D**

#### **Called by:**

JSR at E9CE/EA5D in Move Screen Line.

#### **Entry requirements:**

(D1) should point to the destination screen line. (AC) should point to the original screen line.

#### **Exit conditions:**

(F3) points to color memory for the destination screen line.  
(AE) points to color memory for the original screen line.

#### **Operation:**

1. JSR EA24/EAB2 to set (F3) to point to the color memory location for (D1).
2. Set (AE) to point to the color memory location for (AC). The low byte of the address is the same. The high byte of the

address is obtained by masking off all but the two lowest bits then OR'ing with \$D8/\$94, the value for the starting page of color memory.

### Test for Character Set Switch EC44/ED21-EC77/ED5A

#### Called by:

JMP at E7D2/E7FD in Main Screen Editor; alternate entry at EC4F/ED30 by JMP at E879/E8C0 in Main Screen Editor.

When entered at EC44/ED21, this routine checks to see if the SHIFT and Commodore keys are held down together. If the key value of \$0E (14) is found, it switches to the lowercase character set, then exits from the screen editor routines. When entered at EC4F, this routine checks for key values of \$8E (switch to uppercase character set), \$08 (disable character set switching), and \$09 (enable character set switching). If one of these values is found, that action is taken.

#### Operation:

1. Compare the value in the accumulator to \$0E to test for switching to lowercase. If the current key value is not \$0E, branch to step 3.
2. If the value \$0E is found, reset bits in the VIC chip register at D018/9005 to display the lowercase character set, then JMP E6A8/E6DC to exit from the screen editor routines without displaying a character on the screen.
3. EC4F/ED30: Compare the key value in the accumulator to \$8E, the value for switching to uppercase. If the value \$8E is not found, branch to step 5.
4. If \$8E is found, reset bits in the VIC chip register at D018/9005 to switch the display to the uppercase character set. Then JMP E6A8/E6DC to exit from the screen editing routines without displaying a character.
5. Compare the key value in the accumulator with \$08, the value to disable character set switching. If \$08 is not found, branch to step 7.
6. If \$08 is found, set the high bit of location 0291 to 1. This will disable character set switching with the SHIFT-Commodore key combination, since the Keyboard Scan routine will not act on this combination if the value in 0291 >= 128. This does not, however, disable character set switching caused by the \$0E and \$8E key values.

7. Compare the key value in the accumulator to \$09, the value to enable character set switching. If \$09 is not found, branch to a JMP E6A8/E6DC instruction to exit the screen editor without displaying anything on the screen.
8. If \$09 is found, set the high bit of location 0291 to 0. This will enable character set switching with the SHIFT-Commodore key combination, since the Keyboard Scan routine responds to this key combination if the value in 0291 < 128. Exit from the routine with a JMP E6A8/E6DC to leave the screen editor routines without displaying anything on the screen.

### **Return Number of Columns and Rows in Screen E505-E509**

#### **Called by:**

JMP from Kernal SCREEN vector at FFED.

This routine returns the number of rows and columns in the display screen. The 64 screen has 40 columns and 25 rows; the VIC screen has 22 columns and 23 rows.

#### **Operation:**

1. LDX with the number of columns, \$20/\$16.
2. LDY with the number of rows, \$19/\$17.
3. RTS.

### **Read/Plot Cursor Location E50A-E517**

#### **Called by:**

JMP from Kernal PLOT vector at FFF0.

Depending on the value of the status register carry flag on entry to the routine, either read the cursor location or set the cursor location.

#### **Entry requirements:**

Set the carry bit to read the cursor location and return row and column values in the X and Y registers, respectively. Clear the carry bit to move the cursor to a specified location. The X register should contain the line number for the desired cursor position, and the Y register should hold the column number for the desired cursor position.

**Exit conditions:**

X will contain the line the cursor is on. Y will contain the column the cursor is on. If the carry flag is clear on entry, the cursor is positioned based on the X and Y values on exit.

**Operation:**

1. If carry is set, branch to step 6.
2. If the carry is clear, store the X register in D6, the current physical screen line the cursor is on.
3. Store the Y register in D3, the current cursor position within the logical screen line.
4. JSR E56C/E587 to reset the screen line link pointers.
5. LDX D6, the current physical screen line the cursor is on.
6. LDY D3, the cursor position within the logical screen line.
7. RTS.



## **Chapter 8**

---

# **Serial I/O Routines**



# Serial I/O Routines

---

Serial I/O refers to the I/O (input/output) operations that occur over the serial bus. The serial port connects the 64 and VIC to the serial bus. Several I/O devices such as disk drives and printers can be connected to the serial bus. Each I/O device is an *intelligent* device that contains its own microprocessor and ROM control program that allows the device to know how to respond to the commands it receives over the serial bus from the 64/VIC.

During serial I/O, data is transmitted one bit at a time over the serial bus with the series of bits normally representing a byte of data or a command. RS-232 communications also occur in a bit by bit serial manner, but it uses separate lines for transmitting and receiving, while serial I/O only uses one. Two other lines on the serial bus, the serial clock line and the serial attention line, are also used during serial I/O. Within the 64 and VIC, the data and clock lines are both divided into separate lines for incoming and outgoing signals: serial data input, serial data output, serial clock input, and serial clock output. Only a serial attention output line is connected from 64/VIC hardware to the serial bus attention line. The unused serial attention input line is connected to pin 9 of the user port. The serial bus has one additional signal line: serial service request input. However, the Kernal serial I/O routines do not make use of this line.

Another difference between RS-232 and serial I/O is in the number of bits that make up a discrete unit of transmission. With RS-232, a unit can be less than eight bits, while in serial I/O a unit is always eight bits. Each byte of serial data is transmitted and received from the low bit to high bit direction. For example, ASCII A with a bit value of 0100 0001 is transmitted in this sequence: 10000010.

The tables below show which CIA data port bits are used in serial I/O for the 64 and which VIA data port bits are used in serial I/O for the VIC.

## Serial I/O Routines

### Commodore 64 Serial Port CIA Map

DD00: CIA #2 Data Port A

Bit 3 Serial attention output

Bit 4 Serial clock output

Bit 5 Serial data output

Bit 6 Serial clock input

Bit 7 Serial data input

DC0D: CIA #1 Interrupt Control Register

Bit 4 FLAG IRQ—serial service request input (not used by Kernal serial I/O routines)

### VIC-20 Serial Port VIA Map

911F: VIA #1 Data Port A (without handshaking)

Bit 0 Serial clock input

Bit 1 Serial data input

Bit 7 Serial attention output

9120: VIA #2 Data Port B

CB1 Serial service request input (not used by Kernal serial I/O routines)

CB2 Serial data output

9121 VIA #2 Data Port A (with handshaking)

CA2 Serial clock output

The following table shows the serial I/O line functions as viewed from the serial port.

### Serial Port I/O Lines

Pin	Function
1	Serial service request input
2	Ground
3	Serial attention input/output
4	Serial clock input/output
5	Serial data input/output
6	Reset

The serial I/O lines are connected to all serial devices and each serial device should have a unique address. The serial I/O lines are active low (meaning that the lines remain at +5 volts while inactive, and bus activity is indicated by pulling a line to zero voltage). Any serial device (including the 64/VIC) can bring the clock or data lines low. The serial attention line is used to tell serial devices that a command is coming. No incoming signals on the attention line of the serial port are recognized by the CIA/VIA chip—there's no serial attention

input line connected to the CIA/VIA—so any attempt by another serial bus device to bring the serial attention line low will not be acknowledged by the 64/VIC. The 64/VIC is the only device that controls the serial attention line. Thus, the 64/VIC is the only controller of the serial bus. The TALK, LISTEN, UNTALK, and UNLISTEN commands are sent from the 64/VIC to one particular serial device at a time. When the serial attention line is brought low by the 64/VIC, the serial devices should then prepare for a command to arrive over the serial bus. The first five bits (the low five bits) of the command contain the address of the serial device to which this command is directed, and the last three bits specify the actual command to the serial device. With five bits, 32 possible serial devices can be addressed. However, see the caution by Michael G. Peltier in the *1541 Single Drive Floppy Disk Maintenance Manual* about the maximum number of devices (five) that should be connected to the serial bus at any one time.

Serial I/O can be performed either through BASIC commands or by calling Kernal routines. The Kernal serial I/O routines are described later in this chapter. No examples of serial I/O programs are included here. See the chapter on “Using Disk Storage” in Raeto Collin West’s *Programming the Commodore 64* and *Programming the VIC* for some examples of using machine language programs for serial disk I/O.

Serial I/O is not interrupt-driven like I/O to tape or to an RS-232 device. Tape I/O is performed by the specific tape IRQ interrupt handling routines, and RS-232 I/O occurs from within the NMI interrupt handler. Instead of using interrupts to determine when to send data out or to sample data in, the Kernal serial I/O routines use interrupts (from timer B on CIA #1/timer 1 on VIA #2) to detect timeouts during the attention-response handshake when a device is not present, to know when to perform the EOI handshake during serial receive, to detect read timeouts, and to detect timeouts during the frame handshake at the end of each byte sent or received. So serial I/O operates on the theory that everything is going to proceed in a timely fashion, that the serial device is going to respond within a prescribed time period, and the interrupts are used to enforce these time limits. During these detections of timeouts and timing of EOI, IRQ interrupts are disabled on the 64/VIC. Thus the timer B/timer 2 interrupt that occurs just sets the interrupt flag register at DC0D/912D to indicate a timer

B/timer 2 timeout has occurred. The serial routines that check for timeouts check the interrupt flag in DC0D/912D rather than letting the actual IRQ interrupt from the CIA/VIA occur. Indeed, the IRQ Interrupt Handler routine has no check for these timer B/timer 2 interrupts. Thus, the use of interrupts is quite different with serial I/O than it is with tape or RS-232 I/O.

Kernal Jump Table routines that specifically send or receive data to or from the serial bus (ACPTR, CHRIN, CHROUT, CIOUT) on a byte-by-byte basis, or LOAD/SAVE routines that handle multiple byte transfers, are used to initiate and control serial I/O.

Although both the 64 and VIC *Programmer's Reference Guides* state that bringing the serial service request (SRQ) line low results in the 64/VIC servicing the device that brought SRQ low, no Kernal routines check for this condition. Of course, you could write your own routine to enable SRQ interrupts and then handle any that occurred.

Another interesting item in serial I/O is that CHRIN from a serial device seems to function almost like ACPTR. CHROUT and CIOUT are also very similar. One difference between the routines is that a filename is only sent during the OPEN sequence. Another difference between these routines is that by using the OPEN sequence you are limited to secondary addresses 0-15 and device numbers 4-30. By using the TALK, TKSA, ACPTR or LISTEN, SECOND, CIOUT sequences, it should be possible to use secondary addresses from 0-31 and device numbers 0-30. The following tables compare some of the serial input and output routine sequences available on the 64/VIC.

### **Serial Kernal Jump Table Input Routine Similarities**

#### **TALK-TKSA-ACPTR Sequence**

TALK: JSR ED09/EE14 to send \$4x (TALK) to the serial device.

TKSA: JSR EDC7/EECE to send a secondary address command and do TALK-LISTEN turnaround.

ACPTR: JMP EE13/EF19 to get the byte from the serial data input line.

#### **OPEN-CHKIN-CHRIN Sequence**

OPEN: JSR ED0C/EE17 to command the current serial device to LISTEN; JSR EDB9/EEC0 to send \$Fx command, the secondary address for OPEN, and send the filename; JMP EDFE/EF04 to send an UNLISTEN command to the serial device.

## Serial I/O Routines

**CHKIN:** The X register contains the logical file number at entry. JSR ED09/EE14 to send the TALK command to the serial device. JSR EDC7/EECE to send secondary address and do the TALK-LISTEN turnaround. Store BA in 99, the current input device.  
**CHRIN:** If location 99 contains a value greater than 3, this is a serial device. If the device is not present, it returns \$0D. JMP EE13/EF19 to get byte from the serial data input line.

### Serial Kernal Jump Table Output Routine Similarities

#### **LISTEN-SECOND-CIOUT Sequence**

**LISTEN:** JMP ED0C/EE17 to send \$2x (LISTEN) to the serial device.  
**SECOND:** JMP EDB9/EEC0 to send a secondary address command to the serial device.

**CIOUT:** JMP EDDD/EEE4 to send the buffered character on the serial data output line.

#### **OPEN-CHKOUT-CHROUT Sequence**

**OPEN:** JSR ED0C/EE17 to command the current serial device to LISTEN; JSR EDB9/EEC0 to send \$Fx command, the secondary address for OPEN, and send the filename; JMP EDFE/EF04 to send an UNLISTEN command to the serial device.

**CHKOUT:** The X register contains the logical file number at entry. JSR ED0C/EE17 to send the LISTEN command to the serial device. JSR EDB9/EEC0 to send the secondary address. Store BA in 9A, the current output device.

**CHRIN:** If 9A contains a value greater than 3, this is a serial device. JMP EDDD/EEE4 to send the buffered character on the serial data output line.

While the serial attention line is held low, bytes sent are considered to be commands. The 64 and VIC use the command values shown in the table below.

### Serial Commands

\$2x	001d	dddd	LISTEN
\$3F	0011	1111	UNLISTEN all devices
\$4x	010d	dddd	TALK
\$5F	0101	1111	UNTALK all devices
\$6x	011s	ssss	Secondary address
\$Ex	1110	aaaa	Secondary address for CLOSE
\$Fx	1111	aaaa	Secondary address for OPEN
\$F1	1111	0001	SAVE memory to serial device*
\$F0	1111	0000	LOAD memory from serial device*

aaaa = Secondary address (0-15) for OPEN/CLOSE

s ssss = Secondary address (0-31)

d dddd = Device address (0-30) for LISTEN and TALK. Device 31 is pre-empted by use for UNLISTEN and UNTALK.

## Serial I/O Routines

\* As the 1541 disk drive manual states, channel numbers (secondary addresses) 0 and 1 are reserved for operating system loads and saves, 2–14 are available, and 15 is the error channel.

As the above table indicates, you cannot send UNTALK or UNLISTEN commands just one serial device: these commands are sent to all serial devices. The secondary address can range from 0–31 (decimal). Notice also that there are no specific OPEN or CLOSE commands. Instead OPEN and CLOSE are combinations of commands. OPEN is a LISTEN (possible devices 4–30), followed by the \$Fx OPEN secondary address, followed by the filename, followed by an UNLISTEN to all devices. CLOSE is a LISTEN (possible devices 4–30), followed by the \$Ex CLOSE secondary address. Notice that OPEN for any device in its final step automatically causes all serial devices to unlisten. It seems that only devices 0–30 are valid for LISTEN and TALK, and 4–30 for OPEN, because device 31 is preempted for indicating the UNLISTEN and UNTALK commands, and devices 0–3 are screened by the 64/VIC Kernal routines for OPEN to indicate the keyboard, tape, screen, and RS-232 devices. The TALK and LISTEN routines do not screen for device numbers 0–3. Notice that you can use a secondary address of 0–31 for TKSA and SECOND, while you are limited to 0–15 for OPEN and CLOSE.

The Kernal serial routines also apparently prevent secondary addresses  $\geq 128$  decimal, \$80, from being sent for serial OPEN, LOAD, or SAVE.

## Where to Get More Information

Serial I/O is a topic that could easily fill an entire book. This chapter discusses the Kernal serial I/O routines, but doesn't get into specific details about disk drives and printers. The Commodore 1541 disk drive is extensively covered in *Inside Commodore DOS* by Immers and Neufeld and in *The Anatomy of the 1541 Disk Drive* from Abacus Software, which contains an interesting program that allows spooling of files directly from the disk to the printer while your 64/VIC is busy doing other things. The program essentially commands the printer to listen and the disk to talk. The hardware aspects of the 1541 are covered in the *1541 Single Drive Floppy Disk Maintenance Manual* by Michael G. Peltier. Also see the chapters about disk I/O in Raeto Collin West's *Programming the VIC* and *Programming the Commodore 64*, both from COMPUTE! Books. The



Commodore *VIC-1541 User's Manual* and *VIC-1525 User's Manual* are also occasionally useful. Also, the *Commodore 64 Programmer's Reference Guide* contains diagrams and timings of serial I/O functions on pages 364–65.

The following table illustrates how the various values for the serial attention, clock, and data lines may be set or read. These values indicate the actual setting of the line to high or low at the serial data port.

### Reading and Setting the Serial Clock, Data, and Attention Lines

Line	Logic Value at Serial Port	Where Set or Read: 64	VIC
Attention Out	High = False = 1	DD00 bit 3 = 0	911F bit 7 = 0
	Low = True = 0	DD00 bit 3 = 1	911F bit 7 = 1
Clock In	High = False = 1	DD00 bit 6 = 1	911F bit 0 = 1
	Low = True = 0	DD00 bit 6 = 0	911F bit 0 = 0
Data In	High = False = 1	DD00 bit 7 = 1	911F bit 1 = 1
	Low = True = 0	DD00 bit 7 = 0	911F bit 1 = 0
Clock Out	High = False = 1	DD00 bit 4 = 0	CA2 control = 110
	Low = True = 0	DD00 bit 4 = 1	CA2 control = 111
Data Out	High = False = 1	DD00 bit 5 = 0	CB2 control = 110
	Low = True = 0	DD00 bit 5 = 1	CB2 control = 111

In examining the serial I/O routines, just looking at the code without referring to other sources is not very enlightening. Jim Butterfield's article "How the VIC/64 Serial Bus Works" (*COMPUTE!*, July 1983, pages 178–84) helps make the serial I/O logic comprehensible. Also, Raeto Collin West's *Programming the PET/CBM* (like subsequent volumes on the VIC and 64) clarifies the active low principle in which true is low (0), while false is high (1).

The serial I/O routines perform various handshaking sequences between the controller (the 64/VIC), the talker (either the 64/VIC or a serial device), and the listeners (either the 64/VIC or serial devices). Whenever the controller needs to send a command to a serial device, it brings the serial attention line low. The actual transmission of a byte of data and the handshaking sequences that occur for the data byte transfer are discussed in this chapter; see the Send Serial Byte: Command or Data routine and Figure 8-3. The EOI handshake sequence is also covered there, and illustrated in Figure 8-2. The EOI sequence when the 64/VIC is the listener is also covered in the Receive Byte from Serial Device routine. The sequence called the TALK–LISTEN turnaround that converts the 64/VIC into a listener and a serial device into the talker is described in Figure 8-4.

## **Bring Serial Bus Attention Line High EDBE/EEC5-EDC6/EECD**

### **Called by:**

Falls through from EDBB/EEC2 in Send Secondary Address After LISTEN, JSR at EDD0/EEED7 in Send Secondary Address After TALK and Do TALK-LISTEN Turnaround, JSR at EF03/EF09 in Send UNLISTEN Command, JSR at F281/F33A in Open Serial Output Channel.

This routine brings the serial bus attention line high. The serial attention output line from the CIA/VIA chip passes through an inverter before reaching the serial port. Thus, setting the CIA/VIA pin low (0) brings the serial attention line high at the serial port.

No separate routine exists for the converse function of bringing the serial attention output line low. However, see the code at EDF3/EEF9 in Send UNTALK Command for a sample of how to bring the serial attention output line low.

### **Operation:**

64: AND the contents of the CIA register at DD00 with \$F7 and store the result back into DD00, thus turning off bit 3 in CIA #2 data port A. Bit 3 is the serial attention output line from the CIA chip.

VIC: AND the contents of the VIA register at 911F with \$7F and store the result back into 911F, thus turning off bit 7 in VIA #1 data port A. Bit 7 is the serial attention output line from the VIA chip.

## **Bring Serial Bus Data Line High EE97/E4A0-EE9F/E4A8**

### **Called by:**

JSRs at ED24/EE2E and ED3A/EE43 in Do Attention Handshake with Serial Device, JSRs at ED41/EE4A and ED7A/EE88 in Send Serial Byte: Command or Data, JSR at EE2A/EF26 in Receive Byte from Serial Device, JMP at EE10/EF16 in Send UNLISTEN Command.

This routine brings the serial bus data line high. The serial data output line from the CIA/VIA chip passes through an inverter before reaching the serial port. Thus setting the CIA/VIA pin low (0) brings the serial data line high at the serial port.

### Operation:

1. 64 :LDA DD00 (CIA #2 data port A). VIC: LDA 912C (VIA #2 peripheral handshaking control register).
2. AND \$DF (binary 1101 1111) to turn off bit 5.
3. 64: STA DD00 to set bit 5 of the port, serial data out, to 0.  
VIC: STA 912C to hold the CB2 handshaking line low. Data port bit 5 of CIA #1 of the 64 or the CB2 line of VIA #2 of the VIC then passes through an inverter to reach the serial port data output line.

### Bring Serial Bus Data Line Low EEA0/E4A9-EEA8/E4B1

#### Called by:

JSR at ED75/EE83 in Send Serial Byte: Command or Data, JSR at EDCD/EED4 in Send Secondary Address After TALK and Do TALK-LISTEN Turnaround, JSRs at EE47/EF45 and EE76/EF75 in Receive Byte from Serial Device.

This routine brings the serial bus data line low. The serial data output line from the CIA/VIA chip passes through an inverter before reaching the serial port. Thus setting the CIA/VIA pin high (1) brings the serial data line low at the serial port.

#### Operation:

1. 64: LDA DD00 (CIA #2 data port A). VIC: LDA 912C (VIA #2 peripheral handshaking control register).
2. ORA \$20 (binary 0010 0000) to turn on bit 5.
3. 64: STA DD00 to set bit 5 of the port, serial data out, to 1.  
VIC: STA 912C to hold the CB2 handshaking line high.

### Bring Serial Bus Clock Line High EE85/EF84-EE8D/EF8C

#### Called by:

JSR at ED2B/EE35 in Do Attention Handshake with Serial Device, JSRs at ED49/EE53 and ED7D/EE8B in Send Serial Byte: Command or Data, JSR at EDD3/EEDA in Send Secondary Address After TALK and Do TALK-LISTEN Turnaround, JSR at EE0D/EF13 in Send UNLISTEN Command, JSR at EE18/EF1E in Receive Byte from Serial Device, JSR at FE2E in System Reset (VIC).

## Serial I/O Routines

This routine brings the serial bus clock line high. The serial clock output line from the CIA/VIA chip passes through an inverter before reaching the serial port. Thus, setting the CIA/VIA pin low (0) brings the serial clock line high at the serial port.

### Operation:

1. 64: LDA DD00 (CIA #2 data port A). VIC: LDA 912C (VIA #2 peripheral handshaking control register).
2. 64: AND \$EF (binary 1110 1111) to turn off bit 4. VIC: AND \$FD (binary 1111 1101) to turn off bit 1.
3. 64: STA DD00 to set bit 4 of the port, serial clock out, to 0. VIC: STA 912C to hold the CA2 handshaking line low.

### Bring Serial Bus Clock Line Low

EE8E/EF8D-EE96/EF95

### Called by:

JSR at ED37/EE40 in Do Attention Handshake with Serial Device, JSR at ED5F/EE6C in Send Serial Byte: Command or Data, JSR at EDF0/EEF6 in Send UNTALK Command, JSR at FF36 in System Reset (VIC).

This routine brings the serial bus clock line low. The serial clock output line from the CIA/VIA chip passes through an inverter before reaching the serial port. Thus, setting the CIA/VIA pin high (1) brings the serial clock line low at the serial port.

### Operation:

1. 64: LDA DD00 (CIA #2 data port A). VIC: LDA 912C (VIA #2 peripheral handshaking control register).
2. 64: ORA \$10 (binary 0001 0000) to turn on bit 4. VIC: ORA \$02 (binary 0000 0010) to turn on bit 1.
3. 64: STA DD00 to set bit 4 of the port, serial clock out, to 1. VIC: STA 912C to hold the CA2 handshaking line high.

### Read Serial Data In and Serial Clock In

EEA9/E4B2-EEB2/E4BB

### Called by:

JSRs at ED44/EE4D, ED50/EE5A, ED55/EE60, ED5A/EE66, and EDA6/EEAC in Send Serial Byte: Command or Data, JSR at EDD6/EEDD in Send Secondary Address After TALK and

## Serial I/O Routines

Do TALK-LISTEN Turnaround, JSRs at EE1B/EF21 and EE37/EF35 in Receive Byte from Serial Device.

On the 64, this routine first forces CIA #2 data port A, DD00, to stabilize, then loads the accumulator from DD00 and does an ASL. This ASL leaves the bit from the serial data input line in the carry flag of the status register and the bit from the serial clock input line in the high bit of the accumulator.

On the VIC, this routine first forces VIA #1 data port A, 911F, to stabilize, then loads the accumulator from 911F and does a LSR. This LSR leaves the bit from the serial clock input line in the carry flag of the status register and the bit from the serial data input line in the low bit of the accumulator.

The serial data and clock input lines from the serial port to the CIA/VIA chips do not pass through any inverters. Thus, reading a value of 1 indicates the corresponding serial line is high, and reading a value of 0 indicates the corresponding serial line is low.

### **Operation:**

1. Wait for DD00/911F to stabilize by loading its value into the accumulator and comparing this value to the current value in DD00/911F until the two are the same.
2. 64: ASL to shift the bit for the serial data input line into the carry bit and the bit for the serial clock input line into the high bit of the accumulator.

VIC: LSR to shift the bit for the serial data input line into the low bit of the accumulator and the bit for the serial clock input line into the carry bit.

## **Send LISTEN Command to Device**

**ED0C/EE17-ED10/EE1B**

### **Called by:**

JMP from Kernal LISTEN vector at FFB1, JSR at F648/F6E0 in Send Secondary Address for CLOSE, JSR at F27A/F333 in Open Serial Output Channel, JSR at F3E3/F49F in Send OPEN, LOAD, or SAVE Command to Device, JSR at F60D/F6A5 in Save to Serial Device.

This routine prepares the accumulator, which contains the device number, to send a LISTEN command to the device. The device number should be 0-30. RS-232 interrupts are disabled, then this routine falls through to the Send Serial Control Character routine.

### Operation:

1. The accumulator, which contains the current device number in the low five bits, is ORed with \$20 (binary 0010 0000) to set bits 5-7 to 001 to indicate a LISTEN command.
2. JSR F0A4/F160 to disable RS-232 interrupts.
3. Fall through to ED11/EE1C, which is the routine to send a serial command.

### Do Attention Handshake with Serial Device ED11/EE1C-ED3F/EE48

#### Called by:

Fall through from ED10/EE1B after Send TALK Command to Device or Send LISTEN Command to Device, JSR at EE00/EF06 in Send UNTALK Command or Send UNLISTEN Command; alternate entry at ED36/EE40 by JSR at EDBB/EEC2 in Send Secondary Address After LISTEN, JSR at EDC9/EED0 in Send Secondary Address After TALK and Do TALK-LISTEN Turnaround.

If the serial deferred flag indicates that a character is buffered to be sent on the serial bus, the end-or-identify (EOI) handshake and the buffered character are sent on the bus before the command (control character) is sent. Also, the serial deferred flag and the EOI flag are then turned off.

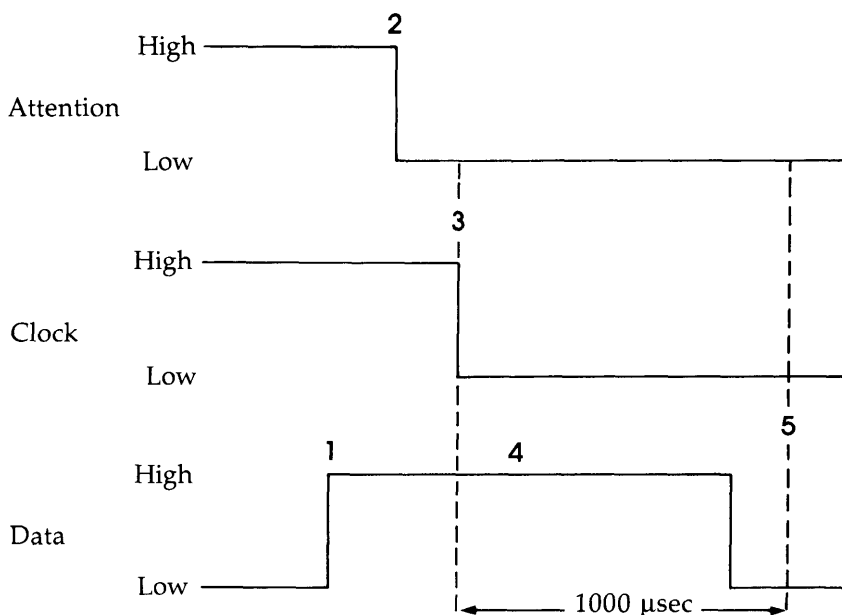
The command in the accumulator at entry, which had been temporarily saved on the stack, is stored in 95, the serial buffered character. This buffered character is used later in the Send Serial Byte routine when it is sent as a command with the serial attention line held low.

Next, the attention request sequence in Figure 8-1 is performed by the controller (the 64/VIC) to inform the serial devices that a command is coming.

If the routine is entered at the alternate point, ED36/EE40, from the routines to send a secondary address after TALK or LISTEN, only steps 3, 4, and 5 in Figure 8-1 are performed, since the serial attention line should still be low.

Once step 5 in Figure 8-1 is finished, this routine falls through to the Send Serial Byte routine at ED40/EE49 where the Send Serial Byte routine expects to find that a serial device has responded to the attention request by bringing the serial data line low. If it doesn't find this condition, it assumes the device is not present.

**Figure 8-1. Attention Request from 64/VIC**



1. Allow serial data line to go high.
2. Bring serial attention line low.
3. Bring serial clock line low.
4. Allow serial data line to go high.
5. Delay one millisecond. Listening devices must bring the data line low within this period to be recognized as present on the bus.

One interesting question about the routine on the VIC is why the delay of one millisecond is done with IRQ interrupts enabled (the SEI is not until EE49). Although it probably is unlikely an interrupt would occur between the one millisecond delay and the following instruction, it is possible and could force a false device-not-present condition. Indeed, on the 64 this oversight is corrected by inserting the SEI before the one millisecond delay and also before bringing the clock line low and allowing the data out to go high.

If this routine is entered from falling through from the serial send TALK command or the serial send LISTEN command, the TALK or LISTEN command is placed in the serial buffered character location to be sent on the serial data output

line. Bringing the serial attention line low causes all devices on the serial bus to listen for a command. Valid device addresses for the command are 0–30. The listening devices on the serial bus can check the first five bits of the command to see if they are the device to which this command is being sent. If the device number is 31, all serial devices are being addressed for an UNLISTEN or UNTALK command. The serial device that recognizes its address then reads the command from the three high bits of the command.

For the alternate entry points for sending a secondary address after LISTEN or TALK, the serial device should still be listening under attention and should read a secondary address of 0–31 from the lower five bits and read the secondary address command identifier in the three high bits.

### Operation:

1. If the serial output deferred flag, 94, has its high bit off, branch to step 5.
2. If the serial output deferred flag, 94, has its high bit on, the Send Serial Byte Deferred routine at EDDD/EEE4 has been executed to set this flag. The flag indicates a character is waiting to be sent. If the serial device is currently listening, the flag will be set. Set the high bit of the end-or-identify (EOI) flag, A3, to 1.
3. JSR ED40/EE49 to send to the serial bus the EOI handshake and the buffered character.
4. Clear the high bits of 94 and A3 to indicate no character is awaiting transmission and the EOI handshake is not to be done.
5. Store the value from the accumulator on entry to this routine in 95, the serial buffered character.
6. 64: Disable IRQ interrupts.
7. JSR EE97/E4A0 to bring the 64/VIC serial data output line high. This will allow the serial bus data line to go high.
8. The BNE at ED29/EE33 is an unconditional branch because the previous JSR EE97/E4A0 ANDs the accumulator with \$DF (binary 1101 1111). Thus, a value of \$3F (0011 1111) could never remain in the accumulator upon return from the JSR, and hence the CMP \$3F is never equal to the accumulator. Both the 64 and the VIC have this apparent bug. It is unclear what condition Commodore was try-



- ing to check for with this comparison for \$3F, but if a match would have been found, a JSR EE85/EF84 to allow the serial clock line to go high would have been executed.
9. Set the 64/VIC serial attention output line low to bring the serial bus attention line low. On the 64, store a 1 in bit 3 of DD00, the serial attention output line. On the VIC, store 1 in bit 7 of 911F, the serial attention output line. Remember that the lines from the CIA/VIA chip go through an inverter before reaching the serial port.
  10. ED36/EE40: Disable IRQ interrupts on the 64 only. JSR EE8E/EF8D to bring the serial clock output line low.
  11. JSR EE97/E4A0 to bring the 64/VIC serial data output line high, allowing the serial bus data line to go high.
  12. JSR EEB3/EF96 to delay one millisecond (1000 microseconds). On the VIC, VIA #2 timer A is used to generate an interrupt after counting one millisecond. On the 64, a delay loop of instructions that takes one millisecond to execute is used.
  13. Fall through to the Send Serial Byte: Command or Data routine at ED40/EE49.

### **Send Serial Byte: Command or Data ED40/EE49-EDAC/EEB3**

#### **Called by:**

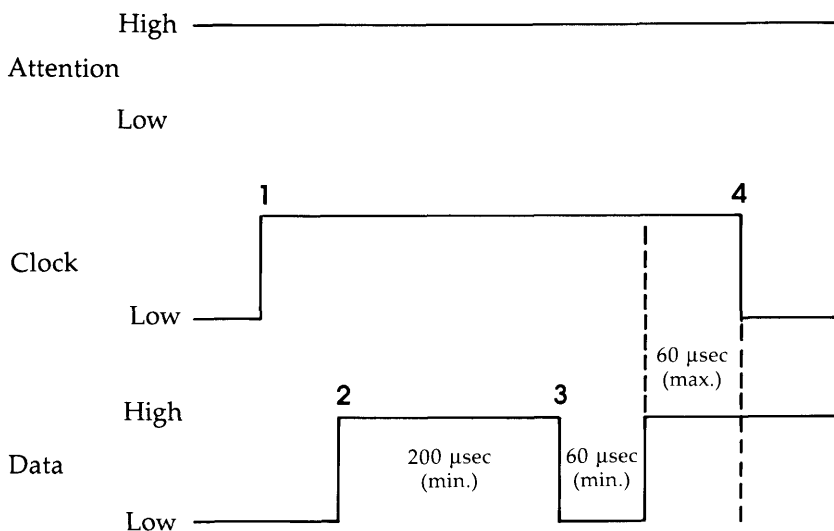
JSR at ED19/EE24 and fall through from ED3D/EE46 in Do Attention Handshake with Serial Device, JSR at EDE7/EEEE in Send Serial Byte Deferred.

This routine is called to send either a command or a data byte on the serial data output line. The data output line is from bit 5 of DD00 CIA #2 data port A on the 64, and from the CB2 handshaking line of VIA #2 port B on the VIC.

The 64/VIC first makes sure that it is not holding the data line low. The routine then tests the data line, and if it is not low, the addressed serial device is not responding and is considered not present.

If the data line is low, the device has responded and a byte can be sent to it. If the EOI flag, A3, has its high bit on, indicating this will be the last data byte to be sent, the EOI handshake shown in Figure 8-2 is performed before sending the buffered character.

## Figure 8-2. EOI Handshake Sequence

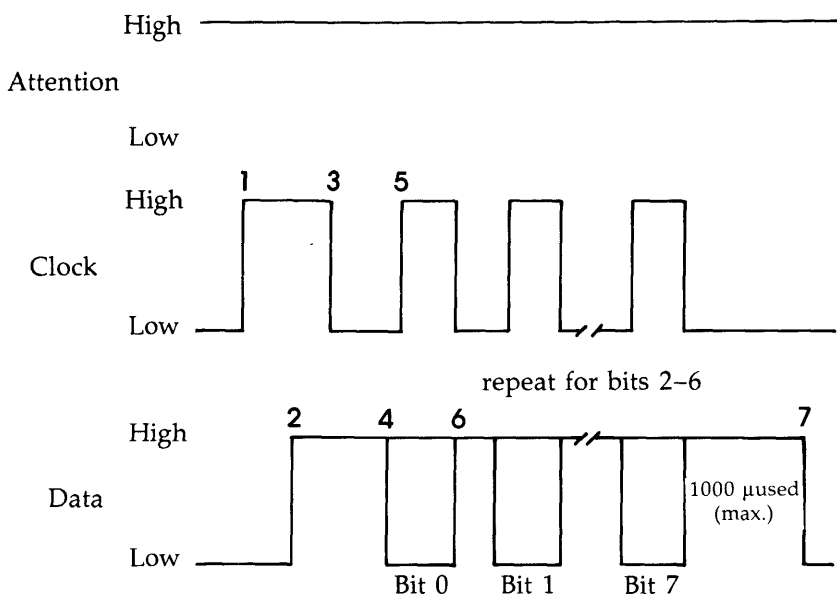


1. The talker allows the clock line to go high when ready to send.
2. The listener allows the data line to go high when ready to receive.
3. If the talker has not brought the clock line low within 200  $\mu\text{sec}$ , the listener assumes this is an EOI handshake. To confirm this, the listener brings the data line low again for 60  $\mu\text{sec}$ , then brings it high again.
4. To confirm the EOI handshake, the talker must not bring the clock line low within 60  $\mu\text{sec}$ .

Rather than the talker bringing the clock line low as it does when it is ready to transmit normal bytes of data, the talker just loops waiting for the data line to go high. The data line goes high when the listener is ready to receive data. The listener can hold off indefinitely allowing the data line to go high until it is finally ready to receive data. The listener must monitor the clock line, and if the talker does not bring the clock line low within 200 microseconds, it is notifying the listener that the EOI sequence is to be performed and that the next byte is the last byte of the file. The listener confirms the EOI sequence by bringing the data line low for at least 80 microseconds (for an external listener) or 60 microseconds (if the 64/VIC is the listener), then allowing the data line to go high again. This routine for the 64/VIC watches for this sequence of data line low, data line high to complete the EOI

handshake. Then the final data byte is transmitted in normal fashion, as shown in Figure 8-3.

**Figure 8-3. Serial Byte Transmission and Handshake**



1. The talker allows the clock line to go high when ready to send.
2. The listener allows the data line to go high when ready to receive. If the talker does not respond within 200  $\mu$ sec, perform EOI handshake (see Figure 8-2).
3. The talker brings the clock line low to signal the start of transmission.
4. The talker sets the data line low for a bit value of 0 or high for bit value of 1.
5. The talker then brings the clock line high to alert the listener that a valid bit is present on the data line. The listener must read the data line before the talker pulls the clock line low again to set the next bit value.
6. The talker brings the clock line low and the data line high between bits.
7. After all eight data bits have been sent, the listener must bring the data line low within 1000  $\mu$ sec to assure the talker that the byte was successfully received.

The talker brings the clock line low after the listener has signaled it is ready to receive data by bringing the data line high. The talker brings the clock line low within 200 micro-seconds if the EOI handshake is not to be done. If the EOI

handshake has been sent, the talker brings the clock line low within 60 microseconds after the listener has brought the data line high to complete the EOI handshake. When the talker brings the clock line low, either for EOI or non-EOI, the talker is ready to send the actual bits for this byte of data.

The routine then sets the serial byte data transfer counter, A5, to 8. The byte is sent one bit at a time from low to high. The data output line is set high for a bit value of 1 and low for a bit value of 0. Each bit has a set up time of around 70 microseconds during which time the clock line is held low by the talker. Once the talker is ready to send the bit, it releases the clock line to high to signal to the listener that the listener should sample the data line for the value of this bit. The listener typically samples the data line within 20 microseconds. However, if the listener is the 64, the 6567 VIC-II chip's cycle stealing for video matrix access or sprite access requires that a minimum of 60 microseconds be allowed for the 64 to sample the data line. Between each bit the data line should be high and the clock line held low. If the serial data input line is low between bits, this routine branches to the read/write timeout status routine.

For each bit that is sent, the serial data transfer counter, A5, is decremented. It continues sending bits until all eight bits have been sent or unless a read/write timeout occurs.

Once the A5 counter reaches zero, indicating all eight bits have been sent, the computer sets CIA #1 timer B/VIA #2 timer 2 to \$0400. Then it loops, waiting for the serial data input line to go low. The serial data input line is brought low by the listener to indicate it has accepted the byte. If the serial data input line does not go low within this countdown of \$0400 (1000 microseconds), the timer B/timer 2 interrupt that occurs indicates a read/write timeout called a frame error for this byte. If the serial data input line goes low within 1000 microseconds, the listener has accepted the data. In this acceptance case, the computer will enable IRQ interrupts and RTS.

During this routine IRQ interrupts are disabled. When IRQ interrupts are disabled, any IRQ interrupt that occurs will not be serviced by the IRQ interrupt handler. However, by checking the flag of the interrupt status register, you can still detect if a timer B/timer 2 interrupt occurred. By this method, even though IRQ interrupts are disabled, interrupts can still occur and be serviced.

## Operation:

1. Disable IRQ interrupts.
2. JSR EE97/E4A0 to bring the serial data output line high. This will allow the serial bus data line to go high.
3. JSR EEA9/E4B2 to sample the status of the serial data input line. If the serial data input line has not been brought low by the listener, BCS to EDAD/EEB4 to set the status word, 90, to indicate a device not present error condition.
4. JSR EE85/EF84 to bring the serial clock output line high.
5. If A3, the EOI flag, has its high bit off, branch to step 8. If the high bit is on, perform the EOI handshake (see Figure 8-2) as this byte to be sent will be the last data byte for a file.
6. Loop until the serial data input line is brought high by the listener.
7. Loop until the serial data input line is brought low by the listener.
8. Loop until the serial data input line is brought high by the listener.
9. JSR EE8E/EF8D to set the serial clock output line low.
10. Set the serial byte data transfer counter, A5, to \$08.
11. Force DD00 (CIA #2 data port A)/911F (VIA #1 data port A) to stabilize.
12. If the serial data input line is low (0) between data bits, branch to EDB0/EEB7 to set read/write timeout status.
13. ROR 95, the serial buffered character, rotating the low bit into the carry.
14. If the carry is clear after the ROR, JSR EEA0/E4A9 to send a 0 by holding the serial data output line low.
15. If the carry is set after the ROR, JSR EE97/E4A0 to send a 1 by holding the serial data output line high.
16. JSR EE85/EF84 to set the serial clock output line high. Then delay for four NOP instructions (about eight microseconds). When you combine the time required for these NOP instructions with the time for the following LDA DD00/912C, AND \$DF, ORA \$10/\$02, STA DD00/912C, you see that the serial clock output line is held high for about 20 cycles. These 20 cycles (20 microseconds) give the listener time to sample the data line.
17. Use the instructions just mentioned in step 16 to bring the serial clock output line low and the serial data output line high.

18. Decrement the serial byte transfer counter, A5.
19. If all the bits in this byte are not yet sent (if A5 is not 0), branch to step 11 to check for a timeout between bits (and to send the next bit if no timeout occurred).
20. After all eight bits have been sent, set CIA #1 timer B/VIA #2 timer 2 to \$0400 to set a delay of approximately 1 millisecond or 1000 microseconds. For the VIC, storing the \$04 in 9129, the high byte of the latch value for timer 2, will load the counter from the latch values, clear the VIA interrupt register, and start the timer countdown. However, for the 64 you must separately order timer B to load its counter from the latches and start counting by storing \$19 into DC0F, the control register for timer B. Also, it is necessary to LDA DC0D to clear any pending timer B interrupts.
21. If a timer B/timer 2 interrupt occurs (determined by looking at DC0D/912D, the interrupt flag register), branch to EDB0/EEB7, the Set Status Word routine, to indicate a frame error condition.
22. If the serial data input line stays high, loop to step 21.
23. If the listener brings the serial data line low before the timer B/timer 2 interrupt occurs (i.e., within one millisecond), then it is acknowledging that the byte has been received, so CLI and RTS.

### **Send OPEN, LOAD, or SAVE Command to Device F3D5/F495-F408/F4C6**

#### **Called by:**

JSR at F37F/F43F in OPEN Execution, JSR at F4C8/F56A in Load or Verify from Serial Device, JSR at F605/F69D Save to Serial Device.

For OPEN, LOAD, or SAVE operations to a serial device, the current device is commanded to listen, then the secondary address and the filename are sent on the serial bus. Finally, all devices on the serial bus are commanded to unlisten.

The secondary address must be less than 128 (decimal) and the filename must contain at least one character.

Before sending the secondary address, this routine ORs the secondary address with \$F0. Thus, only secondary addresses of 0-15 are valid for OPEN, LOAD, or SAVE. A secondary address of 0 indicates a LOAD operation, and a

secondary address of 1 specifies a SAVE. Secondary addresses 2–15 are available for OPEN.

### Entry requirements:

B9 should hold the current secondary address. B7 should hold the number of characters in filename. BA should hold the current device number. (BB) should point to the current filename. (These values can be established using the Kernal SETLFS and SETNAM routines.)

### Operation:

1. If B9, the current secondary address,  $\geq$  \$80 (128 decimal), branch to step 12. Only secondary addresses  $<$  \$80 (128) are acceptable for OPEN, LOAD, or SAVE.
2. If there are no characters in the filename, branch to step 12. B7 contains the number of characters in the filename.
3. 64: Store 0 in 90 to clear the I/O status indicator.
4. LDA BA, the current device number.
5. JSR ED0C/EE17 to send a LISTEN command to the current device. The serial attention output line is brought low to send the LISTEN command, and it will remain low upon return.
6. LDA B9, the secondary address, then ORA \$F0 to prepare to send the secondary address for OPEN, SAVE, or LOAD.
7. JSR EDB9/EEC0 to send the secondary address to the device. The attention line is still low from step 5 when the subroutine is called. However, the serial attention output line will be set high upon return. Thus, the following filename is sent as regular bytes of data, not as a command. (This is contrary to the information about how filenames are sent as commands in the article "How the VIC/64 Serial Bus Works" by Jim Butterfield, *COMPUTE!* 1983.)
8. Test bit 7 of 90, the I/O status word. If bit 7 is set to 1, the preceding subroutine detected that the specified device is not present. In this case, pull the current return address from the stack so that the JMP to F707/F78A to display the DEVICE NOT PRESENT error message will RTS to the routine that called for the OPEN, SAVE, or LOAD.
9. If the device is present, see if there are any characters in the filename. If not, branch to step 11.

10. If the filename contains characters, go through a loop that gets the next character in the filename and outputs this character on the serial bus until all characters in the filename have been transmitted. JSR EDDD/EEE4 to send each character as a byte of data.
11. JMP F654/JSR EF04. At F654 (on the 64), do a JSR EDFE. These instructions on the 64 and the VIC command all devices on the serial bus to unlisten.
12. CLC and RTS.

### **Send Secondary Address After LISTEN EDB9/EEC0-EDBD/EEC4**

#### **Called by:**

JMP from Kernal SECOND vector at FF93, JSR at F286/F33F Open Serial Output Channel, JSR at F3EA/F4A6 in Send OPEN, LOAD, or SAVE Command to Device, JSR at F612/F6AA in Save to Serial Device, JSR at F651/F6E9 in Send Secondary Address for CLOSE.

This routine stores the secondary address passed in the accumulator in 95, the serial buffered character, and does a JSR to ED36/EE40 to send the secondary address to the serial bus as a command. Finally, it falls through to EDBE/EEC5 to bring the serial attention output line high, the setting for transmitting normal data bytes.

#### **Entry requirements:**

The accumulator should hold the secondary address to be sent.

#### **Operation:**

1. STA 95, saving the secondary address in the serial buffered character location.
2. JSR ED36/EE40 to bring the serial clock output line low to indicate the talker is ready to send another byte, delay one millisecond, and send the character in 95 to the serial data output line. The serial attention output line should be set low when calling the subroutine, so that the character in 95, the secondary address, will be considered to be a serial command.
3. Fall through to the routine at EDBE/EEC5 to bring the serial attention output line high.



## **Send Serial Byte Deferred** **EDDD/EEE4-EDEE/EEF5**

### **Called by:**

JMP from Kernal CIOUT vector at FFA8, JSR at F3FE/F4BA in Send OPEN, LOAD, or SAVE Command to Device, JSRs at F61C/F6B4, F621/F6B9, and F62B/F6C3 in Save to Serial Device, JMP at F1D8/F288 in Determine Output Device.

This routine to send a character to the serial bus maintains a one-byte buffer, 95. The character to be sent is stored in this buffer.

This routine first tests a flag, 94, which indicates whether the buffer, 95, already contains a character. If the buffer contains a character, first it sends the buffered character in 95 to the serial bus. Then it stores the current byte in the buffer, 95. If the buffer is empty at entry, it simply stores the character to be sent in the buffer, 95.

### **Operation:**

1. See if the high bit of the serial deferred flag, 94, is set. If so, branch to step 3.
2. If no character is in the buffer, set the high bit of 94 by setting the carry and rotating the carry into bit 7 (ROR 94). Then branch to step 6.
3. Push the byte to be buffered onto the stack.
4. JSR ED40/EE49 to send the byte in 95, the serial deferred character, over the serial data output line.
5. Pull the byte to be buffered from the stack.
6. Store the byte in 95, the serial deferred character.
7. CLC and RTS.

## **Set Status Word** **EDAD/EEB4-EDB8/EEBF**

### **Called by:**

BCS at ED47/EE51 in Send Serial Byte: Command or Data; alternate entry at EDB0/EEB7 by BCC at ED6F/EE7D and BNE at EDA4/EEAA in Send Serial Byte: Command or Data; alternate entry at EDB2/EEB9 by JMP at EE44/EF42 Receive Byte from Serial Device.

Set 90, the I/O status word, to indicate an error condition. The condition indicated depends on the entry point:

## Serial I/O Routines

EDA0/EEB4: device not present.  
EDB0/EEB7: read or write timeout.  
EDB2/EEB9: read timeout.

Clear the carry, enable IRQ interrupts, bring the serial attention output line high, the serial clock line output line high, and the serial data output line low.

### Operation:

1. EDAD/EEB4: LDA \$80 and fall through to step 3 by using a dummy BIT instruction.
2. EDB0/EEB7: LDA \$03.
3. EDB2/EEB9: JSR FE1C/FE6A to set the I/O status word with ORA 90, STA 90.
4. Enable IRQ interrupts and clear the carry.
5. Branch to EE0E/EF09 to bring the serial attention output line high, enter a short delay loop, bring the serial clock output line high, and bring the serial data output line low.

### Delay One Millisecond EEB3/EF96-EEBA/EFA2

#### Called by:

JSR at ED3D/EE46 in Do Attention Handshake with Serial Device.

For the 64, a series of instructions is executed that takes approximately one millisecond.

For the VIC, timer 2 of VIA #2, the timer count, is initialized to \$0400. From this initial value, timer 2 takes approximately one millisecond to countdown to zero and generate an interrupt.

### Operation:

For the 64:

1. TXA (2 cycles).
2. LDX \$B8 (2 cycles).
3. DEX (2 cycles  $\times$  184 times = 368 cycles).
4. BNE to step 3 (3 cycles  $\times$  184 times = 552 cycles).
5. TAX (2 cycles).
6. RTS (6 cycles).

The JSR to call this routine also takes 6 cycles. Thus, for the 64 the total number of cycles is  $6 + 2 + 2 + 368 + 552 + 2 + 6 = 938$  cycles, which when divided by the 64's clock

## Serial I/O Routines

frequency of 1,022,370 cycles per second is 917 microseconds, or approximately one millisecond.

For the VIC:

1. LDA \$04.
2. STA 9129, VIA #2 high byte of timer 2 counter. When this STA is done, the timer 2 interrupt flag for VIA #2 is cleared, the low latch of the counter is transferred to the low counter, and the counter begins decrementing at the system clock rate.
3. LDA 912D, VIA #2 interrupt flag register.
4. AND \$20.
5. BEQ to step 3 as long as the timer 2 flag has not been set.
6. RTS.

A timer 2 value of \$0400 counts down at the system clock rate. \$0400 = 1024 decimal, and 1024 cycles divided by 1,022,370 cycles per second is 1001 microseconds, approximately 1 millisecond.

### **Save to Serial Device F5FA/F692-F641/F6D9**

#### **Called by:**

Falls through from Determine Device for SAVE routine at F5ED/F685.

To save to a serial device, a filename must be specified. If no filename is given, jump to the error routine to display the MISSING FILE NAME error message and exit.

Send the LISTEN command to the current serial device, send the secondary address command of \$61 to indicate a SAVE operation. Then, if the device is present, send all the characters in the filename. If the device is not present, exit with the DEVICE NOT PRESENT error message.

Set the pointer to the current byte to save, (AC), from the starting address of the memory to be saved. This starting address is then sent over the serial bus, first AC and then AD. The address in (AC) is incremented after each byte is sent to the serial device. When the address in these bytes equals the address in (AE), the pointer to the end of the memory being saved + 1, the save is complete.

When the save is complete, command all serial devices to unlisten and fall through to the routine at F642/F6DA to send

the secondary address for a CLOSE command to the serial device.

The actual operation for saving each byte is to LDA with the next byte from the save area, using (AC) as a pointer, and then send this byte over the serial bus. The routine also checks to see if the keyboard STOP key has been pressed, and if it has, the save is halted and the routine falls through to the routine to send the CLOSE command to the device. If the STOP key has not been pressed, (AC), the pointer to the save area, is incremented. If the high byte of the pointer is \$00, save is halted and the routine falls through to the routine to send the CLOSE command. Thus, you can't wrap your save from FFFF to 0000.

### Entry requirements:

B7 should hold the number of characters in filename. BA should hold the current device number. (C1) should point to start of the save area. (AE) should point to the end of the save area + 1.

### Operation:

1. LDA \$61 and STA B9 to set the secondary address to \$61 indicating a SAVE.
2. If the filename does not contain any characters (if B7 contains a 0), JMP F710/F793 to display the MISSING FILE NAME error message, set the accumulator to 8, set the carry bit to 1, and exit.
3. JSR F3D5/F495 to send the LISTEN command, send the secondary address of \$61, send all the characters in the filename, and then send the UNLISTEN command.
4. JSR F68F/F728 to display SAVING and the filename.
5. JSR ED0C/EE17 to send the LISTEN command to the current device, BA. Then JSR EDB9/EEC0 to send the secondary address of \$61 to the device.
6. JSR FB8E/FBD2 to set the pointer to the start of the save area, (AC), from (C1). (C1) was set from two bytes in page zero during the Jump to SAVE Vector routine (see chapter 5).
7. Send the low byte of the starting address of the save area, AC, to the serial bus by JSR EDDD/EEE4.
8. Send the high byte of the starting address of the save area, AD, to the serial bus by JSR EDDD/EEE4.
9. JSR FCD1/FD11 to compare (AC) to (AE) to see if (AC), the pointer to the save area, is greater than or equal to the pointer to (AE), the end of the save area + 1.

10. If (AC)  $\geq$  (AE), branch to step 16.
11. LDA (AC),Y to get the next byte from the save area. The Y register is zero.
12. Send this byte onto the serial bus with JSR EDDD/EEE4.
13. JSR FFE1 (the Kernal STOP vector) to see if the STOP key on the keyboard has been pressed. If it has, fall through to the routine at F633/F6CB to send the CLOSE command to the serial device, load the accumulator with 0, set carry, and exit.
14. If the STOP key has not been pressed, branch to F63A/F6D2 to skip the CLOSE command, then increment the pointer to the save area, (AC), by JSR FCDB/FD1B.
15. If this pointer's high byte, AD, is 0, fall through to step 16; otherwise, branch to step 9.
16. JSR EDFE/EF04 to send the UNLISTEN command to all serial devices.
17. Fall through to the routine at F642/F6DA to send the LISTEN command to this device, send the secondary address command of \$E1 to indicate a CLOSE for a SAVE. Then command all serial devices to unlisten.

### Stop Load or Save

#### F633/F6CB-F639/F6D1

##### Called by:

JMP at F4FE/F595 in Load or Verify from Serial Device, falls through from F632/F6CA in Save to Serial Device.

This routine is executed if the keyboard STOP key is detected during a load or save for a serial device.

JSR F642/F6DA to do the following: send the LISTEN command to this device, send secondary address command for CLOSE (\$E0 for LOAD, \$E1 for SAVE), send the UNLISTEN command to all devices, clear the carry, and RTS.

After the JSR returns, load the accumulator with 0, set the carry, and exit.

##### Operation:

1. JSR F642/F6DA to send the CLOSE command for SAVE or LOAD (\$E1 or \$E0 respectively), send UNLISTEN, CLC, and RTS.
2. LDA \$00.
3. SEC.
4. RTS.

## **Send Secondary Address for CLOSE** **F642/F6DA-F658/F6F0**

### **Called by:**

Falls through from F641/F6D9 in Save to Serial Device, JSR at F2EE/F3AE in Close Logical File for Serial Device, JSR at F52B/F5C2 in Load or Verify from Serial Device, JSR at F633/F6CB in Stop Load or Save; alternate entry at F6F4 by JMP at F406 in Send OPEN, LOAD, or SAVE Command to Device (64).

If the current secondary address < \$80 (128 decimal), send the LISTEN command to the serial device, send the secondary address for a CLOSE command of \$Ex (*x* varies depending on what called this routine), send the UNLISTEN command to all serial devices, clear carry, and RTS.

If the current secondary address ≥ \$80 (128 decimal), just CLC and RTS.

### **Operation:**

1. If the current secondary address ≥ \$80, branch to step 6.
2. JSR ED0C/EE17 to send the LISTEN command to the current serial device.
3. LDA B9, the current secondary address. Then AND \$EF (1110 1111 binary) to clear bit 4 and ORA \$E0 to set bits 5–7 to 1. Thus, the command for CLOSE is produced—\$Ex, where *x* is the secondary address option. The *x* will be 0 for a CLOSE following LOAD, 1 for a CLOSE following SAVE, and 2–15 for a CLOSE following OPEN.
4. JSR EDB9/EEC0 to send the secondary address command to the current device.
5. JSR EDFE/EF04 to command all serial devices to unlisten.
6. CLC and RTS.

## **Load or Verify from Serial Device** **F4B8/F55C-F532/F5C9**

### **Called by:**

Falls through from F4B7/F55B in Determine Device for LOAD.

When loading from a serial device, you must specify a filename.

The routine displays SEARCHING FOR and the filename. It sends the LISTEN command to the current serial device, a secondary address of \$60 indicating a LOAD, and the file-

name. It then sends the UNLISTEN command to all serial devices.

Next, it sends the TALK command to the current serial device and the current secondary address of \$60, and then receives a byte from the serial bus. Sending the TALK secondary address command of \$60 performs the TALK–LISTEN turn-around sequence where the serial device becomes the talker and the 64/VIC becomes the listener. If the I/O status word indicates the byte was not returned fast enough, a read timeout has occurred and FILE NOT FOUND is displayed.

The first two bytes received from the serial device are used as the pointer to the starting address of the load/verify area in (AE). However, if you set the secondary address to 0 and call the load routine, the X and Y registers at entry to load are used to set the pointer to the start of the load/verify area in (AE). After initializing (AE), the routine displays the message for LOADING or VERIFYING.

Next, the following loop is executed until the status word, 90, indicates EOI (end of file):

- Turn off the status word read error bit.
- See if the STOP key is down. If so, send the LISTEN command to the serial device, the CLOSE command secondary address of \$E0, and the UNLISTEN command, then exit.
- Receive a byte from the serial device; bytes being received now are actual data values loaded into memory (or verified against memory).
- If the current operation is a VERIFY, compare the byte received against the byte pointed to by (AE). If no match, set the verify mismatch bit in the status word.
- If the current operation is a LOAD, store the byte received in memory location pointed to by (AE).
- Increment the pointer to the load/verify area, (AE).

Once the EOI status is received from the serial device, the load/verify is considered complete and (AE) points to the end of the load area + 1. Any bytes sent by the device after it has indicated EOI are discarded by the 64/VIC. For EOI, the 64/VIC sends the UNTALK command to all serial devices, which thus forces the serial device to send its last buffered character. After UNTALK, the LISTEN command is sent to the serial device, followed by the CLOSE secondary address of \$E0, and the UNLISTEN command to all serial devices.

The final location of the pointer (AE), indicating the end of the load area + 1, is loaded into X from AE and into Y from AF, and the routine exits.

### Entry requirements:

B7 should hold the number of characters in filename. (C3) should point to the starting address for LOAD. 93, the LOAD/VERIFY flag, should indicate the operation: 0 for LOAD or 1 for VERIFY. 90, I/O status word, should have a value of 0.

### Operation:

1. If there are no characters in the filename, JMP F710/F793 to display the MISSING FILE NAME message, set the carry, set the accumulator to 8, and exit.
2. 64: LDX with the current secondary address, B9. VIC: JSR to a patch area at E4BC to LDX B9.
3. JSR F5AF/JMP F647 (from the patch area at E4BE on the VIC) to display SEARCHING FOR and the filename.  
Set the current secondary address, B9, to \$60.
4. JSR F3D5/F495 to send the LISTEN command to the current device, the secondary address of \$60, the characters of the filename, and to send the UNLISTEN command to all serial devices.
5. JSR ED09/EE14 to command the current device, BA, to talk.
6. JSR EDC7/EECE to send the current secondary address in B9 of \$60 and to do the TALK–LISTEN turnaround. Now the serial device is the talker, and the 64/VIC (and possibly other serial devices) is the listener.
7. JSR EE13/EF19 to receive a byte from the serial bus, with the byte received returned in the accumulator.
8. STA AE, since the first byte received should be the low byte of the pointer to the end of the load area + 1.
9. If the I/O status word, 90, indicates a read timeout, branch and JMP F704/F787 to display the FILE NOT FOUND error message, set accumulator to 4, set carry, and exit.
10. JSR EE13/EF19 to receive a byte from the serial bus, with the byte received returned in the accumulator .
11. STA AF, since the second byte received should be the high byte of the pointer to the end of the load area + 1.



## Serial I/O Routines

12. 64: See if the secondary address specified when the load was called is 0. This secondary address was saved in the X register at entry to this routine. If it is 0, the starting address for the load is taken from the setting of the X and Y registers at the time load was called. If the secondary address is 0, the X and Y values (found now in C3 and C4) are stored in AE and AF. Thus, a secondary address of 0 for a load allows a relocatable load. After this, JSR F5D2 to display the LOADING or VERIFYING message.

VIC: A patch is used for this test of the secondary address. JSR E4C1 to test the secondary address in the same manner as the 64 did, and then JMP F66A to display the LOADING or VERIFYING message.

13. Clear the I/O status word read error bit.
14. JSR FFE1 (the Kernal STOP vector) to test for the STOP key. If the STOP key is detected, JSR F633/F6CB to send a LISTEN command to this serial device, send the CLOSE secondary address command of \$E0, send the UNLISTEN command to all serial devices, LDA \$00, SEC, and exit.
15. JSR EE13/EF19 to receive a byte from the serial device.
16. If the status word, 90, indicates read timeout, branch to step 13. Thus, a read timeout for the individual bytes of data does not abort the entire load or verify.
17. If this is a LOAD operation, branch to step 19.
18. If this is a VERIFY operation, see if the last byte received from the serial device is the same as the byte pointed to by (AE). If not, set the verify mismatch bit in the I/O status word with LDA \$10, JSR FE1C/FE6A. Then use a dummy BIT instruction to skip to step 20.
19. For a LOAD operation, store the byte received from the serial device at the current location pointed to by (AE).
20. Increment (AE), the pointer to the load/verify area.
21. See if the I/O status word indicates EOI. If not, branch to step 13.
22. If EOI has been detected, JSR EDEF/EEF6 to send the UNTALK command to all serial devices, JSR F642/F6DA to send the LISTEN command to this serial device, send the CLOSE secondary address command of \$E0, and send the UNLISTEN command to all serial devices.
23. Clear the carry, LDX from AE, LDY from AF, and RTS.

## **Send TALK Command to Device ED09/EE14-ED10/EE1B**

### **Called by:**

JMP from Kernal TALK vector at FFB4, JSR at F238/F2F1 in Open Serial Input Channel, JSR at F4CD/F56F in Load or Verify from Serial Device.

The accumulator, which contains the device number, is ORed with \$40, turning on bit 6 to indicate a TALK command. RS-232 interrupts are disabled. The routine then falls through to ED11/EE1C, the Do Attention Handshake with Serial Device routine, to send the serial TALK command.

### **Operation:**

1. ORA \$40. Use a BIT instruction to fall through to step 2, bypassing the Send LISTEN Command to Device routine's entry point at ED0C/EE17.
2. JSR F0A4/F160 to disable RS-232 interrupts.
3. Fall through to the routine to send a serial command at ED11/EE1C.

## **Send Secondary Address After TALK and Do TALK-LISTEN Turnaround EDC7/EECE-EDDC/EEE3**

### **Called by:**

JMP from Kernal TKSA vector at FF96JSR at F245/F2FE in Open Serial Input Channel, JSR at F4D2/F574 Load or Verify from Serial Device; alternate entry at EDCC/EED3 by JSR at F23F/F2F8 in Open Serial Input Channel.

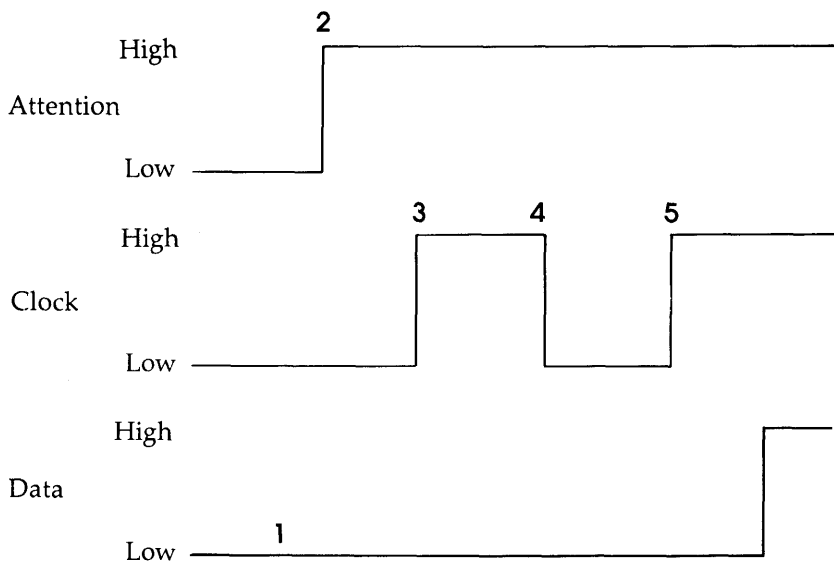
For the normal entry point, store the secondary address in the serial buffered character location at 95 and send the buffered character as a command (with the serial attention line held low) on the serial bus. From this point on, the normal and alternate entry points perform the same functions.

Disable IRQ interrupts. Then perform the TALK-LISTEN turnaround sequence shown in Figure 8-4.

Once this turnaround is complete, the device is now the talker and the 64/VIC is the listener. The talker is holding the clock low and allowing the data to go high. The listener (64/VIC) is holding the data line low.

After these operations, the talker brings the clock line high to indicate it is ready to send data, just as the 64/VIC

**Figure 8-4. TALK-LISTEN Turnaround Sequence**



1. After the desired device has been commanded to talk, the 64/VIC brings the data output line low. This causes no immediate change because the device will already be holding the data line low.
2. The 64/VIC brings the attention line high.
3. The 64/VIC brings the clock line high and waits for the clock line to go low again.
4. To acknowledge the turnaround, the serial device must bring the clock line low and release the data line to go high. The data line will remain low, however, because the 64/VIC is holding its data output line low.
5. The serial device is now the talker and the 64/VIC is now the listener. Transmission of bytes will proceed as shown in Figure 8-3.

does when it is the talker and is ready to send data.

The talker (the serial device) then sends the data to the listener (the 64/VIC and any other listening serial devices) as shown in Figure 8-3.

The routine enables IRQ interrupts and then exits.

If the secondary address  $\geq \$80$  (128 decimal) when opening a serial channel, the secondary address is not sent.

## Entry requirements:

The accumulator should contain the secondary address.

## Operation:

1. STA (the secondary address) in 95, the serial buffered character.

## Serial I/O Routines

2. JSR ED36/EE40 to send the secondary address as a command on the serial data output line.
3. EDCC/EED3: Disable IRQ interrupts.
4. JSR EEA0/E4A9 to hold the serial data output line low.
5. JSR EDBE/EEC5 to bring the serial attention output line high.
6. JSR EE85/EF84 to bring the serial clock output line high.
7. JSR EEA9/E4B2 to read the serial clock input line. Loop until the serial clock input line is brought low by the serial device.
8. Enable IRQ interrupts and RTS.

### **Receive Byte from Serial Device** **EE13/EF19-EE84/EF83**

#### **Called by:**

JMP from Kernal ACPTR vector at FFA5, JSRs at F4D5/F577, F4E0/F582, and F501/F598 in Load or Verify from Serial Device, JSR at F1B5/F26C in Get Character from Serial Input Channel.

In this routine, the serial device is the talker, while the 64/VIC is the listener.

IRQ interrupts are disabled, and the serial byte data transfer counter is initialized to 0.

The serial clock output line is brought high. Next, loop until the serial clock input line goes high, and then set the serial data output line low.

Set a timer interrupt for 250 microseconds. If the serial clock input line does not go low within the 250 microseconds, perform the EOI handshake, which consists of bringing the serial data output line low, the serial clock output line high, then the serial data output line high.

Set the EOI status bit in the I/O status word, 90. Check again to see if the serial clock input line goes low within 250 microseconds. If it doesn't, set the read timeout status. If it does, the listener is now ready to receive data.

If the serial clock input line goes low before the first 250-microsecond delay has completed, the EOI sequence is not executed; the 64/VIC is ready to receive data.

For receiving the byte, loop until the serial clock input line goes low and receive eight bits from the serial data input line. As the data comes in, a series of shifts builds the serial byte received into location A4. Between bits, the serial clock output line goes low.

## Serial I/O Routines

After all eight bits have been received, bring the serial data output line low. If the EOI status is true, send the UNLISTEN command to all serial devices. Exit with the accumulator containing the byte received, the carry clear, and IRQ interrupts enabled.

Although IRQ interrupts are disabled during this routine, the timer interrupt can occur and still be detected by looking at the interrupt flag register in the CIA/VIA.

### Operation:

1. Disable IRQ interrupts.
2. Set the serial byte data transfer counter, A5, to 0.
3. JSR EE85/EF84 to set the serial clock output line high. This allows the serial bus clock line to go high.
4. JSR EEA9/E4B2 to read the serial clock input line. Loop until the serial clock input line is brought high by the serial device, indicating it is ready to send.
5. 64: Set DC07, the high byte of CIA #1 timer B, to \$01, preparing it for a timer count of \$0100. This will generate an interrupt in about 250 microseconds. Store \$19 in DC0F to load the timer B counter from the latched value and to start timing.

VIC: JSR E4A0 to bring the serial data output line high indicating to the serial device that the VIC is ready to receive data.

6. 64: JSR EE97 to bring the serial data output line high indicating to the serial device that the 64 is ready to receive data. Clear any pending interrupts in CIA #1 with LDA DC0D.

VIC: Set 9129, the high byte of VIA #2 timer 2, to \$01 to specify a timer count of \$0100. This will generate an interrupt in about 250 microseconds.

7. See if a timer B/timer 2 interrupt occurs by examining the flag bit for the timer in the interrupt flag register, DC0D/912D. If an interrupt has occurred, branch to step 10.
8. If no timer B/timer 2 interrupt occurs, JSR EEA9/E4B2 to read the serial clock input line. If the serial clock input line is still high, branch to step 7. If the serial clock input line is low, continue with step 9.
9. When the serial clock input line goes low before a timer B/timer 2 interrupt, the serial device is indicating that the EOI handshake should not be performed for the byte to follow, so branch to step 16.

## Serial I/O Routines

10. When a timer B/timer 2 interrupt occurs before the serial clock input line is brought low, the serial device is assumed to be calling for an EOI handshake. If A5, the transfer counter, is 0, branch to step 12.
11. If A5 is nonzero, this is the second time an interrupt has occurred while the clock line is high. Thus, the serial device has not responded to the EOI handshake from the 64/VIC. Set the read timeout status with LDA \$02, JMP EDB2/EEB9, which also exits from this routine.
12. JSR EEA0/E4A9 to bring the serial data output line low for the EOI handshake.
13. 64: JSR EE85 to allow the serial clock output line to remain high.

VIC: JSR EF0C. At EF0C first enter a delay loop for about 52 microseconds, then JSR EF84 to allow the serial clock output line to remain high and JMP E4A0 to bring the serial data output line high.

Both the 64 and VIC acknowledge the EOI by bringing the data output line low for at least 60 microseconds and then releasing the data line to high. The VIC performs the data low/data high operations in steps 12 and 13, while the 64 performs the data low/data high operations in steps 13 and 6.

14. Set the EOI status bit in 90, the I/O status word, with LDA \$40 and JSR FE1C/FE6A. Increment A5, which will now be used to indicate that if another timer B/timer 2 interrupt occurs, it is a timeout and not an EOI.
15. Loop to step 5/step 6.
16. When the serial has indicated that it is ready to send the byte, set A5, the serial byte data transfer counter, to 8.
17. Read and stabilize the data port for the serial data input line, DD00 (CIA #2 port A)/91FF (VIA #1 port A).
18. 64: ASL to move the value of the serial clock input line to the high bit of the accumulator and the value of the serial data input line into the carry.

VIC: LSR to shift the value of the serial clock input line into the carry and the value of the serial data input line into the low bit of the accumulator.

19. Loop to step 17 until the serial clock input line goes high. It is brought high by the serial device (the talker in this case) when the device has completed the setup of the data line and valid data exists. The valid data that exists when

the clock line is brought high has already been read by the 64/VIC in step 17.

20. VIC: LSR to shift the serial data input line value into the carry for the following ROR.
21. ROR A4 to move the serial bit just received (now in the carry bit) into the high bit of A4, the serial byte being built.
22. Again read and stabilize DD00/911F.
23. 64: ASL to move the serial clock input line value to the high bit of the accumulator.  
VIC: LSR to shift the serial clock input line value into the carry.
24. Loop to step 22 until the serial clock input line goes low. When it does, the data is no longer considered valid as the talker is doing the setup for the next bit.
25. Decrement A5, the serial byte transfer counter.
26. If A5 is not 0 after the decrement, branch to step 17 to read the next bit of data.
27. If A5 is now 0, all eight bits of serial input have been received. JSR EEA0/E4A9 to bring the serial data output line low to indicate to the talker that the 64/VIC has accepted this byte.
28. 64: Check 90, the I/O status word to see if the EOI status bit is set. VIC: Check 90 to see if any I/O status bits are set. If the status is not flagged, branch to step 30.
29. If EOI (64) or any I/O status word condition (VIC) is indicated, JSR EE06/EF06 to first execute a delay loop of approximately 50–60 microseconds, JSR EE85/EF84 to bring the serial clock output line high, and then JMP EE97/E4A0 to bring the serial data output line high.
30. LDA A4 to exit with the accumulator containing the serial byte that was built from the eight bits of serial data received.
31. Enable IRQ interrupts, CLC, and RTS.

### Open Serial Input Channel F237/F2F0–F24F/F308

#### Called by:

BCS at F221/F2DA in CHKIN Execution.

The current device (with a device number  $\geq 4$ ) is commanded to talk.

## Serial I/O Routines

If the current secondary address < \$80 (128 decimal), the secondary address is sent on the serial bus. However, a secondary address that  $\geq$  \$80 (128 decimal) is not sent.

If the serial device is not present, the DEVICE NOT PRESENT message is displayed. If the serial device is present, the current device number, BA, is stored as the current input device number, 99.

### Entry requirements:

The accumulator should hold the device number. B9 should hold the current secondary address.

### Operation:

1. TAX to preserve the accumulator value in X, then JSR ED09/EE14 to send a TALK command to the serial device whose device number was passed in the accumulator .
2. If the secondary address  $\geq$  \$80, no secondary address is sent. Instead, just JSR EDCC/EED3 to do the TALK-LISTEN turnaround, then JMP to step 4.
3. If the current secondary address < \$80, JSR EDC7/EECE to send the secondary address and do the TALK-LISTEN turnaround. Return with the attention line set high.
4. TXA to retrieve the device number value saved in step 1, then check bit 7 of 90, the I/O status word, which indicates the device-not-present condition. If the specified serial device is not present, JMP F707/F78A to exit with the DEVICE NOT PRESENT error message, set the carry, and set accumulator to 5.
5. If the serial device is present, BPL to F233/F2EC to set the input device number, 99, from the current device number in the accumulator.
6. CLC and RTS.

## Get Character from Serial Input Channel F1AD/F264-F1B7/F26E

### Called by:

BCS at F173/F22A in Determine Input Device.

If any I/O status errors occur, load the accumulator with \$0D (ASCII carriage return) and exit from the routine.

If no I/O status errors occur, JMP EE13/EF19 to receive a byte from the serial device. Exit with the byte in the accumulator.



### Operation:

1. If any I/O status errors are indicated in 90, LDA \$0D, CLC, and RTS.
2. If no I/O status errors are indicated, JMP EE13/EF19 to receive a byte from the current serial device. Exit with the carry clear and with the accumulator containing the byte received.

### Open Serial Output Channel F279/F332-F290/F349

#### Called by:

BCS at F266/F31F in CHKOUT Execution.

This routine opens an output channel for a serial device for subsequent CHROUTs to the serial device. It sends a LISTEN command to the serial device and, if a secondary address < 128 (decimal) was specified, the secondary address is also sent.

#### Entry requirements:

The accumulator should contain the device number. B9 should hold the current secondary address.

### Operation:

1. TAX to preserve the accumulator value in X, then JSR ED0C/EE17 to send the LISTEN command to the current serial device.
2. If the secondary address, B9, < \$80 (decimal 128), branch to step 4.
3. If the secondary address >= \$80, JSR EDBE/EEC5 to set the serial attention output line high. Branch to step 5.
4. JSR EDB9/EEC0 to send the secondary address in B9 to the current serial device. Return with the attention line set high.
5. TXA to retrieve the device number saved in step 1, then test the I/O status register, 90. If the high bit of 90 is 1, JMP F707/F78A to display the DEVICE NOT PRESENT error message, set the accumulator to 5, set the carry, and exit.
6. If the device is present, BPL to F275/F32E to set 9A, the current output device number, from the device number value in the accumulator.
7. CLC and RTS.

### **Send UNTALK Command** **EDEF/EEF6-EE12/EF18**

#### **Called by:**

JMP from Kernal UNTALK vector at FFAB, JSR at F340/F400 in Clear Serial Channels and Reset Default Devices, JSR at F528/F5BF in Load or Verify from Serial Device.

This routine performs the sequence necessary to send the UNTALK command to all serial devices.

#### **Operation:**

1. JSR EE8E/EF8D to set the serial clock output line low.
2. Bring the serial attention output line low by storing a 1 in bit 3 of DD00/bit 7 of 911F.
3. LDA \$5F, the command for all devices to untalk. Use a dummy BIT instruction to fall through to EE00/EF06, step 2 of the following routine, Send UNLISTEN Command.

### **Send UNLISTEN Command** **EDFE/EF04-EE12/EF18**

#### **Called by:**

JMP from Kernal UNLSN vector at FFAE, JSR at F339/F3F9 in Clear Serial Channels and Reset Default Devices, JSR at F4C2 in Send OPEN, LOAD, or SAVE Command to Device (VIC), JSRs at F63F/F6D7 and F654/F6EC in Save to Serial Device; alternate entry at EE03/EF09 by BCC at EDB7/EEBE in Set Status Word; alternate entry at EE06/EF0C by JSRs at EF48 (VIC) and EE7D/EF7C in Receive Byte from Serial Device.

This routine performs the sequences necessary to send the UNLISTEN command to all serial devices.

1. LDA \$3F, the command for all devices to unlisten.
2. JSR ED11/EE1C to do the attention handshake and send the command byte in the accumulator over the serial data output line.
3. EE03/EF09: JSR EDBE/EEC5 to bring the serial attention output line high.
4. EE06/EF0C: TXA to preserve the X register value in the accumulator, then LDX with \$0A/\$0B and DEX until it is zero to introduce a delay of approximately 50–60 micro-seconds. After the delay, TAX to restore the X register value.
5. JSR EE85/EF84 to bring the serial clock output line high.

6. JMP EE97/E4A0 to bring the serial data output line high and exit the routine.

### Close Logical File for Serial Device F2EE/F3AE-F2F0/F3B0

#### Called by:

BCS at F2A5/F353 in Determine Device for CLOSE.

If the current secondary address < \$80 (128 decimal), send the CLOSE command to the serial device and command all serial devices to unlisten.

If the current secondary address ≥ \$80 (128 decimal), do not send the CLOSE or UNLISTEN commands.

Fall through to the Common Exit for Close Logical File Routines routine at F2F1/F3B1 (see chapter 5). The common CLOSE routine decrements the number of open files and removes entries for this file from the secondary address, logical file, and device number tables.

#### Operation:

1. JSR F642/F6DA. If the secondary address < \$80 (128 decimal), send the LISTEN command to the current device, convert the secondary address to \$Ex to indicate the CLOSE command and send this secondary address. Then send the UNLISTEN command to all serial devices.
2. Fall through to the common close logical file routine at F2F1/F3B1.

### Set Serial Timeout Value FE21/FE6F-FE24/FE72

#### Called by:

JMP from Kernal SETTMO vector at FFA2.

The IEEE timeout flag, 0285, is set to the value passed in the accumulator. However, although the purpose of this Kernal routine has been described as setting a flag for timeout conditions on the IEEE bus, nowhere in BASIC or the Kernal is this flag read.

#### Operation:

1. STA 0285, the timeout flag.
2. RTS.



## **Chapter 9**

---

# **RS-232 I/O Routines**



# RS-232 I/O Routines

---

The 64 and VIC provide Kernal routines to handle RS-232 input and output (I/O). RS-232 is a recommended standard of the Electronics Industries Association that applies to serial binary data communication (of any character length and code) between a data terminal (DTE) and data communications equipment (DCE). The 64/VIC is the DTE. A modem is the most common DCE, although other devices such as printers can be used as the DCE. (The complete designation of the current standard is RS-232-C, where the C indicates the third revision.)

The 64/VIC does not follow all of the recommendations in the standard. For example, not all of the RS-232 pins are implemented on the 64/VIC, and none of the RS-232 timing signals or secondary data signals are used. The RS-232 signal lines are accessed through the 64/VIC user port, rather than through a standard RS-232 connector. Also, the voltage levels for signals from the 64/VIC are TTL levels (0 to 5 volts) rather than the true RS-232 voltage range in which a transition past -3 volts indicates a logical 1 and a transition past +3 volts indicates a logical 0. RS-232 interface cartridges are available that convert the 64/VIC TTL voltage signals to true RS-232 voltages. Such interfaces are not necessary to use Commodore's own modems with the 64 or VIC, or to use any of the third-party modems designed specifically for the 64 or VIC. However, almost all other standard RS-232 equipment will require the interface to work with the 64/VIC.

For a copy of the EIA RS-232-C standard, contact the Electronic Industries Association, Engineering Department, 2001 Eye Street, N.W., Washington, D.C. 20006, or phone them at (202) 457-4966.

Table 9-1 shows the RS-232 signal lines provided by the 64/VIC.

## Table 9-1. RS-232 Circuits in the 64 and VIC

EIA Circuit Designation	RS-232 Pin	64/VIC User Port Pin	CIA #2/VIA #1 Pin	To/From 64/VIC	Line Modes	Function
AA	1	A	GND	N/A	X 3	Protective Ground
AB	7	N	GND	N/A	X 3	Signal Ground
BA	2	M	PA2/CB2	From	X 3	Transmitted Data
BB	3	B	FLAG/CB1	To	X 3	Received Data
BB	3	C	PB0	To	X 3	Received Data
CA	4	D	PB1	From	X 3*	Request to Send
CB	5	K	PB6	To	X	Clear to Send
CC	6	L	PB7	To	X	Data Set Ready
CD	20	E	PB2	From	X 3*	Data Terminal Ready
CE	22	F	PB3	To		Ring Indicator
CF	8	H	PB4	To	X	Received Line Signal
		J	PB5			Unassigned

\* Indicates held high during 3-line handshaking.

For a complete definition of what each of these RS-232 interchange circuits does, see the EIA standard. For a more thorough introduction to RS-232, see *RS-232 Made Easy* by Martin Seyer.

The 64/VIC Kernal routines have two built-in modes of RS-232 communication: 3-line and x-line. The 3-line mode only uses the Transmitted Data line and the two Received Data lines with the only handshaking being that the Request to Send and Data Terminal Ready lines are held high. The x-line mode performs a handshaking sequence between the 64/VIC and the modem. Other modes are possible, such as using the Ring Indicator, but these are left to the user to implement.

X-line handshaking has potential problems on the VIC due to two checks of the wrong VIA register for testing the status of the Data Set Ready and Clear to Send lines. At F512 in the routine to open an RS-232 device, bit 7 of 9120 is checked to detect Data Set Ready missing, but 9110 should have been checked instead. It so happens that bit 7 of 9120 (used for keyboard column scan) will always be 1, and thus will always show the Data Set Ready signal as being present. Also, at EFF4 when preparing to transmit the next byte, again 9120 is checked, this time for Data Set Ready and Clear to Send. This incorrect check could explain the note in the VIC *Programmer's Reference Guide* about Clear to Send not being implemented. These problems with x-line handshaking should not occur on the 64 since DD01 is correctly checked for Clear to Send and Data Set Ready.



To test Clear to Send or Data Set Ready on the VIC, you must write your own routine to open an RS-232 device (it could be modeled after the Kernal routine at F4C7–F541 with LDA 9120 changed to LDA 9110). You would also need to intercept NMI interrupts to test for timer 1 interrupts (RS-232 transmit) and test for Data Set Ready and Clear to Send before sending the next byte.

The RS-232 operation of the 64/VIC is a close software emulation of the 6551 Asynchronous Communication Interface Adapter (ACIA) chip, including emulation of the ACIA control register at location 0293, the command register at 0294, and the status register at 0297. Table 9-2 below shows the control, command, and status register values.

The conventional way to use RS-232 from machine language is to first call the Kernal routines SETLFS, SETNAM, and OPEN. The device number when calling SETLFS should be 2 for RS-232. If you are using a printer as the RS-232 device, a logical file number greater than or equal to 128 allows line feeds after a carriage return. The secondary address is not used when opening an RS-232 file.

If you are transmitting RS-232 data, first open an output channel for the logical file by using CHKOUT and then use CHROUT to send the individual bytes of data. If you are receiving RS-232 data, first open an input channel for the logical file by using CHKIN and then receive the individual bytes of data using CHRIN.

The actual transmission of RS-232 data is driven by NMI interrupts. CIA #2 timer A/VIA #1 timer 1 interrupts control transmission; FLAG of CIA #2/CB1 of VIA #1 and CIA #2 timer B/VIA #1 timer 2 interrupts control reception of data.

RS-232 operations use two 256 byte (one page) buffers—one for transmitting and one for receiving. These buffers are implemented as circular queues with each capable of actually holding 255 bytes of data. Two pointers to each of the buffers are maintained. The head points to where bytes are removed from the buffer, and the tail points to where bytes are inserted. An empty buffer is indicated when the pointer to the head has the same value as the pointer to the tail; a full buffer is indicated when the indicator to the head is equal to the value of the tail pointer plus one. Figure 9-1 illustrates this concept.

## Table 9-2. RS-232 Control, Command, and Status Registers

### Control Register (0293)

Bits	7	6	5	4	3	2	1	0
	S	W	W	U	B	B	B	B

S = stop bits. 0 = 1 stop bit, 1 = 2 stop bits.

WW = word length. 00 = 8 bits, 01 = 7 bits, 10 = 6 bits, 11 = 5 bits.

U = unassigned.

BBBB = baud rate. 0001 = 50 baud, 0010 = 75 baud, 0011 = 110 baud, 0100 = 134.5 baud, 0101 = 150 baud, 0110 = 300 baud, 0111 = 600 baud, 1000 = 1200 baud, 1001 = 1800 baud, 1010 = 2400 baud. For the 64, 0000 = user defined baud rate; this feature is not implemented on the VIC, where 0000 will produce an invalid baud rate value. For the VIC, 1011 = 3600 baud (despite the indication in the *Programmer's Reference Guide* that this rate is not implemented). 1011 is not implemented on the 64, and baud rates for 1100-1111 are not implemented on either the 64 or VIC.

### Command Register (0294)

Bits	7	6	5	4	3	2	1	0
	P	P	P	D	U	U	U	H

PPP = parity. xx0 = parity disabled, 001 = odd parity, 011 = even parity, 101 = mark parity, 111 = space parity.

D = duplex. 0 = full duplex, 1 = half duplex.

UUU = unassigned.

H = handshake. 0 = 3-line, 1 = x-line.

### Status Register (0297)

Bits	7	6	5	4	3	2	1	0
	B	D	U	C	E	R	F	P

B = break detected.

D = DSR (Data Set Ready) signal missing.

U = Unused.

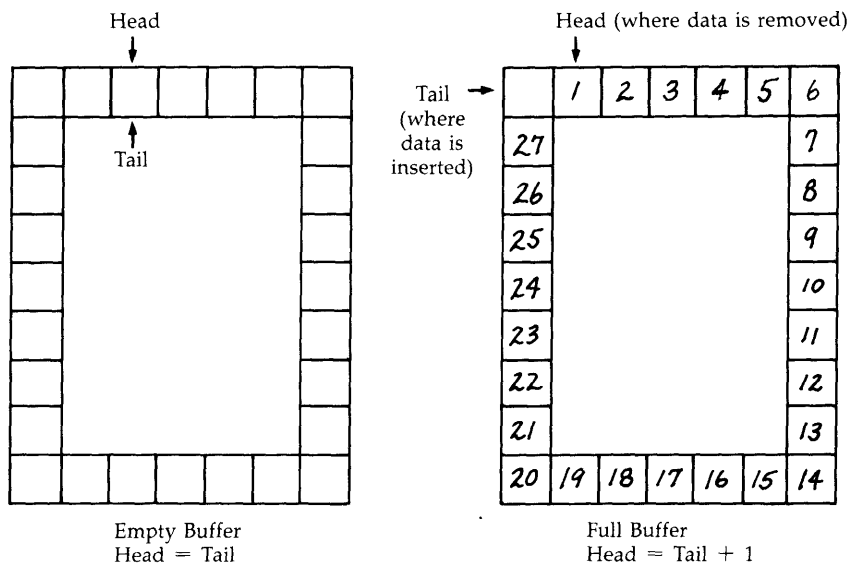
C = CTS (Clear to Send) signal missing.

E = 64: Receive buffer empty; VIC: unused.

R = Receive buffer overrun.

F = Framing error.

P = Parity error.

**Figure 9-1. Circular Transmit/Receive Buffers: Full and Empty**

Whenever the Kernal tries to start serial I/O or tape I/O, the Kernal first checks to see if either timer A/timer 1 or timer B/timer 2 interrupt for CIA #2/VIA #1 is enabled. If so, tape and serial I/O wait for both timer A/timer 1 and timer B/timer 2 interrupts to be disabled. Thus, serial or tape I/O cannot begin if the 64/VIC is currently transmitting or receiving over the RS-232 interface. However, once the serial or tape I/O routines get control and disable the CIA #2 timer A, timer B, and FLAG interrupts (on the 64) or the VIA #1 timer 1, timer 2, and CB1 interrupts (on the VIC), no RS-232 processing can occur until the timer A/timer 1 and FLAG/CB1 interrupts are reenabled in a new RS-232 session.

Also, you cannot load or save to an RS-232 device, since the LOAD and SAVE routines consider an RS-232 device to be invalid.

During RS-232 operations, the interrupt conditions for timer A/timer 1, timer B/timer 2, and FLAG/CB1 of CIA #2/VIA #1 are as follows:

Timer A/timer 1 interrupts (RS-232 transmit) are enabled initially from CHROUT when the first byte is prepared to be sent. Timer A/timer 1 interrupts are disabled only when the RS-232 logical file is closed.

FLAG/CB1 interrupts (RS-232 receive new byte) are enabled by the routine to open an RS-232 channel for input, which is called by the Kernal routine CHKIN. FLAG/CB1 interrupts are also enabled whenever a byte has been received (whether or not in error) and when timer B/timer 2 cannot find a start bit. FLAG/CB1 interrupts are disabled by opening an RS-232 channel for output using x-line handshaking. FLAG/CB1 interrupts are also disabled when the RS-232 logical file is closed, and by the NMI interrupt handler when a FLAG/CB1 interrupt occurs.

The FLAG/CB1 interrupt tells the 64/VIC that a new byte of data is being received at the user port. It has served its purpose then, so it turns itself off, and turns timer B/timer 2 interrupts on to receive each bit of the byte. Then when the byte has been received and it's time to watch for another byte coming along, the timer B/timer 2 interrupts are turned off and FLAG1/CB1 interrupts are reenabled.

Timer B/timer 2 interrupts (RS-232 receive bits) are enabled by the NMI interrupt handler when a FLAG/CB1 interrupt occurs. These interrupts are disabled whenever a byte of data has been received or when timer B/timer 2 cannot find a start bit. Also, timer B/timer 2 interrupts are disabled when the logical RS-232 file is closed.

Timer A/timer 1 interrupts are serviced first by the NMI interrupt handler. If a timer B/timer 2 interrupt or a FLAG/CB1 interrupt should occur while the NMI interrupt handler is servicing the timer A/timer 1 interrupt, the timer B/timer 2 or FLAG/CB1 interrupt is still serviced. The 64 specifically checks for a timer B/timer 2 or FLAG/CB1 interrupt after it finishes servicing a timer A/timer 1 interrupt, but the VIC does not. Instead, it reenables the line settings that allow another NMI interrupt to occur by bringing the NMI line high. If the VIA #1 chip has a pending timer 2 or CB1 interrupt, another execution of the NMI interrupt handler occurs, thus causing nested NMI interrupts. By allowing both the transmit and receive interrupts to be serviced by the NMI interrupt handler no matter when they occur, you can use both CHROUT and GETIN to an RS-232 device without being

concerned over separating the times when you are receiving from the times you are transmitting information.

If you want to simultaneously send and receive via RS-232, your machine language program must open both an input channel for RS-232 and an output channel for RS-232, then do CHROUTs and GETINs. The sequence of doing CHKIN and GETIN or CHKOUT and CHROUT does not seem important when you have selected the 3-line handshaking mode. However, in x-line handshaking, it appears that CHKOUT disables FLAG/CB1 and timer B/timer 2 interrupts, preventing further receiving of input through GETIN. Thus, if you are using x-line handshaking, you should call CHKOUT before calling CHKIN.

Also, see the NMI interrupt handler description since RS-232 interrupts for timer A/timer 1, timer B/timer 2, and FLAG/CB1 are discussed there. One feature that is slightly different on the 64 is the use of a separate byte at 02A1 to indicate which interrupts are enabled for DD0D, CIA #2 interrupt control register. This separate byte is required since the CIA interrupt control register is a write-only mask register and a read-only data register. *Thus, reading DD0D does not show which interrupts are enabled, it only shows which enabled interrupts have actually occurred.*

### **Open Logical File for RS-232 Device F409/F4C7-F482/F541**

#### **Called by:**

JMP at F388/F448 in OPEN Execution; alternate entry at F47D/F53C by JMP at F2C5/F38A in Close Logical File for RS-232 Device.

To prepare for RS-232 communications the routine performs the following steps:

- Sets the data direction register for port B of CIA #2/VIA #1 to make Data Set Ready, Clear to Send, Received Line Signal, Ring Indicator, and Received Data input lines.
- Sets the data direction register for port B of CIA #2/VIA #1 to make Data Terminal Ready and Request to Send output lines and sets the data register to bring these lines high.
- VIC: Sets the CB1 line of port B to produce an interrupt upon a high-to-low transition of CB1.
- Holds CB2 line for Transmitted Data high (1).
- Clears the RS-232 status register.

The baud rate setting in the control register is used to set the value for bit time (the time to transmit one bit). The number of bits to be sent or received is set to 9, 8, 7, or 6 depending on the word length specified in the control register.

Next, the routine determines the type of handshaking requested in the command register. If x-line handshaking is requested, a test for Data Set Ready is performed. However, on the VIC the incorrect use of VIA #2 port B rather than VIA #1 port B for reading the Data Set Ready signal results in this routine always showing that the data set (the modem) is ready, whether or not it actually is.

Finally, two 256-byte buffers, one for receive and one for transmit, are carved out of the top portion of memory, the pointers to the end of memory are reset, and the head and tail of each of these buffers are initialized. The transmit buffer is located one page (256 bytes) lower in memory than the receive buffer. See the comments in this chapter on the GETIN from an RS-232 Device routine to determine how to increase the size of these buffers.

For the VIC, the routine will never indicate that the Data Set Ready signal is missing when x-line handshaking is used. If you need this function, you could copy the routine to a location in RAM (the routine occupies F4C7–F541), change the instruction LDA 9120 to LDA 9110, and finally, change the OPEN vector in (031A) to point to your new location for the routine. However, 9120 is also incorrectly used in the routine at EFEE during RS-232 transmission.

What happens if you try to open additional RS-232 logical files without first closing previously opened RS-232 files? Normally, this routine would find that the high bytes of the pointers to the buffers are not 0, and would exit without allocating new buffers, leading to reusing the same buffers allocated in the initial OPEN. Utter confusion would then result if more than one logical file tried to use the same buffer. It seems possible that you could open more than one RS-232 logical file if you first reset the high bytes F8 and FA to 0. You would probably need to save the (F7) and (F9) pointers to the first logical file buffers so that you could switch between the buffers for the logical files.

**Entry requirements:**

B7 should hold the number of characters in the filename. (BB) should be the pointer to the filename, which contains values

for the command and control registers. (These locations can be set using the Kernal SETNAM routine.) 029B, the tail of the receive buffer, should normally be 0 on entry. 029E, the tail of the transmit buffer, should normally be 0 on entry.

### **Exit conditions:**

0293, RS-232 control register, is set with values for the number of stop bits, word length, and baud rate. 0294, RS-232 command register, is set with values for parity, duplex setting, and handshaking mode. 0299–029A represent the bit time (how long it takes to send a bit) for RS-232 transmission. 0298 holds the number of bits to be sent or received. 029C, the head of the receive buffer, is initialized to the same value as in 029B. 029D, the head of the transmit buffer, is initialized to the same value as in 029E. (F7) is the pointer to the receive buffer base location. (F9) is the pointer to the transmit buffer base location. (0283), the pointer to the end of memory, is reset to a value 512 bytes below its value at entry. Also, the carry bit will be set and the accumulator will contain \$F0 upon exit.

### **Operation:**

1. 64: JSR F483 to initialize CIA #2 register values, then continue with step 3.

VIC: Set VIA #1 port B data direction and initial values as shown below:

#### **Initial RS-232 Line Settings for VIC**

Input lines with initial value of 0:

PB7, Data Set Ready  
 PB6, Clear to Send  
 PB4, Received Line Signal  
 PB3, Ring Indicator  
 PB0, Received Data

Output lines with initial value of 1:

PB2, Data Terminal Ready  
 PB1, Request to Send

2. VIC: Set VIA #1 port B I/O register CB1 line, Received Data, to produce an interrupt upon a high to low transition on CB1. Hold the CB2 line, Transmitted Data, high.
3. Store 0 in the RS-232 status register, 0297.
4. If B7 indicates that there are no characters in the filename, branch to step 6. When opening an RS-232 file, the filename is used to specify values for the control and com-

- mand registers, and, for the 64, to specify a user-defined baud rate.
5. If a filename does exist, store from one to four characters from the name, starting at 0293. The first two characters specify respectively the RS-232 control register, 0293, and the RS-232 command register, 0294. The third and fourth characters are stored in 0295 and 0296, the bit time value. These values are ignored by the VIC, and will only be used by the 64 if the baud rate bits in 0293 are all 0. If only one character is used in the filename, the RS-232 command register, 0294, is not set from the filename. If the command register is 0, it defaults to parity disabled, full duplex, and 3-line handshaking.
  6. JSR EF4A/F027 to compute the number of bits to be sent or received per byte of data. This value is one more than the word length, allowing for a decrement of this value to reach zero when all bits have been sent or received. If the word length specified in the control register is 00, return with a value of 9; if the word length is 01, return with a value of 8; if the word length is 10, return with a value of 7, and if the word length is 11, return with a value of 6. Store the number of bits returned by the subroutine in location 0298, the number of bits to be sent or received.
  7. 64: If the baud rate is specified as 0, don't get any values from the baud rate table. Instead, branch to step 8 to use the values placed in 0295-0296 from the filename (step 5). For nonzero baud rates specified in the lower four bits of 0293, test the PAL/NTSC flag, 02A6, to see whether the PAL baud rate factor table or the NTSC baud rate factor table should be used. This check is necessary because the 64's system clock frequency, which is also the counting frequency for the CIA timers, is different for the two video systems. Take the two bytes for the baud rate factor from the table at FEC2 (NTSC) or E4EC (PAL), and store the bytes in 0295-0296.
  8. 64: Multiply the value in 0295-0296 by two, and add \$C8. Store this value at 0299-029A as the time it takes to transmit or receive one bit. A patch area at FF2E-FF40 is used for part of this calculation. No check is made for the



invalid baud rate values in 0293, so if the lower four bits are 1100–1111, bytes from beyond the table area will be read, resulting in bit timing values that do not correspond to any standard baud rate.

VIC: Use the baud rate specified in the lower four bits of the control register, 0293, to index into the baud rate table at FF5C. Take the two bytes from the baud rate table corresponding to this baud rate, multiply them by two, and add \$C8. Store this value at 0299–029A, as the time it takes to transmit or receive one bit. No check is made that the baud rate value specified in 0293 is valid. Thus, if the lower four bits of that location are 1101–1111, bytes from beyond the table area will be read, resulting in bit timing values that do not correspond to any standard baud rate. Also, the VIC routine does not implement the user-defined baud rate feature, but fails to check for the 0000 bit value in 0293 that indicates this function. Specifying 0000 for the baud rate bits results in the routine reading the bytes at FF5A–FF5B (PLA and RTI opcodes) as the baud rate factor, for a useless rate of approximately 31 baud. A superfluous BNE F4F4 appears in this section of code.

The tables below show the baud rate factor values (in hex) and the corresponding bit times stored in 029A–0299 (also in hex) for all implemented baud rates for the VIC (NTSC) and 64 (PAL and NTSC tables).

#### Baud Rate Tables and Bit Times for VIC

Baud Rate	Bit Time		Table Value	
	029A	0299	High	Low
50	4F	EC	27	92
75	35	48	1A	40
110	24	54	11	C6
134.5	1D	B0	0E	74
150	1A	A4	0C	EE
300	0D	52	06	45
600	06	AA	02	F1
1200	03	54	01	46
1800	02	38	00	B8
2400	01	AA	00	71
3600	01	1C	00	2A

## Baud Rate Tables and Bit Times for 64

Baud Rate	NTSC				PAL			
	Bit Time	Table	Value		Bit Time	Table	Value	
	029A	0299	High	Low	029A	0299	High	Low
50	4A	50	27	C1	4C	FA	26	19
75	35	44	1A	3E	33	50	19	44
110	24	52	11	C5	22	FC	11	1A
134.5	1D	B0	0E	74	1C	98	0D	E8
150	1A	A2	0C	ED	19	A8	0C	70
300	0D	52	06	45	0C	D4	06	06
600	06	A8	02	F0	06	6A	02	D1
1200	03	54	01	46	03	36	01	37
1800	02	38	00	B8	02	24	00	AE
2400	01	AA	00	71	01	9A	00	69

9. If the 3-line handshake method is being used as indicated by bit 0 of the command register, branch to step 11.
10. 64: For x-line handshaking, test bit 7 of DD01, the RS-232 Data Set Ready (DSR) signal. If the DSR bit = 1, continue with step 11. If the DSR bit = 0, JSR F00D to set bit 6 in the status register, 0293, to indicate Data Set Ready missing.

VIC: For x-line handshaking, the value of bit 7 of 9120 is tested. However, as noted earlier, location 9110 should have been used. Thus, the Data Set Ready test for x-line handshaking on the VIC is not performed correctly. It happens that all routines that use 9120 leave the value \$F7 in 9120 at exit. So, this step should always find bit 7 set to 1. The routine never performs the JMP F016 to indicate the Data Set Ready missing error. Instead, it thinks the Data Set Ready signal is present and falls through to step 11.

11. Next, the head of the receive buffer, 029C, is set from the value in 029B, the tail of the receive buffer. The head of the transmit buffer, 029D, is set from the value in 029E, the tail of the transmit buffer. Normally, these values are all 0. If they are not 0, the head and tail pointers still function properly.
12. JSR FE27/FE75 to load the X register with the low byte of the pointer to the end of memory, and the Y register with the high byte of the pointer to the end of memory.
13. If the high byte of the pointer to the RS-232 receive buffer base location, F8, is not 0, branch to step 15. F8 should be

- 0 on the first pass through this routine. It should also be 0 if an RS-232 file has been previously closed.
14. Decrement the Y register, the high byte pointer to the end of memory, to carve out a page (256 bytes) of memory for a receive buffer. Set the pointer to the receive buffer base location, (F7), from the current values in the X and Y registers.
  15. If the high byte of the pointer to the transmit buffer base location is not 0, branch to step 17.
  16. Decrement the Y register, the high byte of the pointer to the end of memory, to carve out a page (256 bytes) of memory for a transmit buffer. Set the pointer to the transmit buffer base location, (F9), from the current values in the X and Y registers.
  17. F47D/F53C: Set the carry, and LDA \$F0.
  18. JMP FE2D/FE7B to set the pointer to the end of memory, (0283), from the current values of the X and Y registers, thus normally two pages (512 bytes) have been subtracted from the top of memory for the RS-232 buffers. Then RTS to exit the routine.

### **CIA Initialization for RS-232 (64) F483-F49D**

#### **Called by:**

JSR at F409 in Open Logical File for RS-232 Device, JSR at F2AF in Close Logical File for RS-232 Device.

This routine initializes the CIA lines used for RS-232 operations on the 64 and also clears 02A1, the byte used to indicate which of the timer A, timer B, and FLAG interrupts are enabled for RS-232 operations.

#### **Operation:**

1. Store \$7F in DD0D, the CIA #2 interrupt control register, to disable all interrupts from CIA #2.
2. Store \$06 in DD03, the CIA #2 data direction register for data port B, and also store \$06 in DD01, CIA #2 data port B. Steps 2 and 3 give the settings shown here:

#### **Initial RS-232 Line Settings For 64**

Input lines with initial value of 0:

PB7, Data Set Ready

PB6, Clear to Send

PB4, Received Line Signal

PB3, Ring Indicator

PB0, Received Data

Output lines with initial value of 1:

PB2, Data Terminal Ready

PB1, Request to Send

3. ORA DD00, port A data register, with \$04 to set PA2, RS-232 Transmitted Data, to 1.
4. Store 0 in 02A1, the RS-232 activity register.
5. RTS.

### Compute Bit Count

**EF4A/F027-EF58/F035**

#### Called by:

JSR at F41D/F4E7 Open Logical File for RS-232 Device.

Bits 6 and 5 of the RS-232 control register, 0293, specify the length of data in each RS-232 character sent or received. This routine uses these two bits to compute the number of bits to be sent or received and returns this number of bits plus one in the X register. Having a value of one greater than the length of a data word allows the transmit and receive routines to decrement this counter each time a bit is sent or received, so that when the counter reaches zero all bits have been sent or received.

#### Entry requirements:

0293, the RS-232 control register, should have bits 6–5 indicating the data length of a character—00 for eight bits, 01 for seven bits, 10 for six bits, and 11 for five bits.

#### Exit conditions:

The X register holds the counter for the number of bits to be sent or received.

#### Operation:

1. Initialize the X register to 9.
2. If bit 5 of the control register is 0, branch to step 4.
3. If bit 5 of the control register is 1, decrement the X register; thus, if the character length is five or seven bits, the X register is now 8.
4. If bit 6 of the control register is 0, branch to step 6.

5. If bit 6 is 1, the X register is decremented twice. Thus, if the character length is six or five bits,  $X = X - 2$ . The X register is now 7 or 6.
6. RTS with the X register set to 9 for 00, 8 for 01, 7 for 10, or 6 for 11.

### **Open RS-232 Channel for Input F04D/F116-F085/F14E**

#### **Called by:**

JMP at F227/F2E0 in CHKIN Execution.

To prepare for RS-232 input, this routine must be executed.

If 3-line handshaking is being used, just see if both timer B/timer 2 interrupts and FLAG/CB1 interrupts are enabled. If either one is enabled, CLC and RTS. If neither one is enabled, enable FLAG/CB1 interrupts, CLC, and RTS.

If x-line handshaking with full duplex is being used, take the same steps.

If using half duplex x-line handshaking, first check to see if the Data Set Ready line is high. If DSR from the RS-232 device is on, also test the Request to Send (from 64/VIC). If DSR is off, branch to indicate the Data Set Not Ready error. If DSR and RTS are both on, wait for any RS-232 output to finish by looping until the CIA #2 timer A/VIA #1 timer 1 interrupt is disabled. Then turn off the Request to Send line, wait for the Data Terminal Ready line to turn on, and enable FLAG/CB1 interrupts for receiving RS-232 data. Finally, CLC and RTS. Since half duplex allows transmission in either direction over the lines, but not in both directions at the same time, this x-line and half-duplex sequence makes certain before receiving data that the 64/VIC is not trying to send any data.

X-line handshaking in the half-duplex mode is performed correctly here for the VIC since the routine uses 9110.

#### **Entry requirements:**

The accumulator must hold 2, the device number for RS-232, upon entry. Bit 0 of location 0294 should be set to 0 for 3-line handshaking or to 1 for x-line handshaking.

#### **Operation:**

1. Set the input device number (at location 99) to 2.
2. If 3-line handshaking is indicated by a 0 in bit 0 of 0294, the RS-232 command register, go to step 3. If x-line handshaking is being used, go to step 4.

3. 64: LDA 02A1, AND \$12, and BEQ to step 10. The AND tests bits 0 and 4 of the RS-232 active interrupt flag byte. Bit 1 is 1 if timer B interrupts are enabled, and bit 4 is 1 if FLAG interrupts are enabled. Thus, branch if neither of these interrupts is enabled.

VIC: LDA 911E, AND \$30, and BEQ to step 10. The AND tests bits 4 and 5 of the RS-232 active interrupt flag byte. Bit 4 is 1 if CB1 interrupts are enabled, and bit 5 is 1 if timer 2 interrupts are enabled. Branch if neither of these interrupts is enabled.

If either timer B/timer 2 or FLAG/CB1 interrupts are enabled, just CLC and RTS, as the interrupts for RS-232 reception are already active.

4. Test bit 4 of 0294, which indicates the duplex mode. If using x-line handshaking and full duplex mode, branch to step 3. If using x-line handshaking and half duplex, fall through to step 5.
5. Test DD01/9110 to see if Data Set Ready line is on. If not, branch to F00D/F0E8 to set the status register, 0297, to indicate that the DSR signal is missing. DSR is an input signal to the 64/VIC from the RS-232 device.
6. Test DD01/9110 to see if Request to Send is off. If it is off, CLC and RTS (the machine language instruction) as no transmission is active. If it is on, continue with step 7.
7. F062 (64): LDA 02A1, LSR, BCS F062. Loop until bit 0 of 02A1 becomes 0, which indicates timer A interrupts for RS-232 transmission have been disabled.

F12B (VIC): BIT 911E, BVS F12B. This code loops until bit 6 of 911E becomes 0, which indicates that timer 1 interrupts for RS-232 transmission are disabled.

8. Turn off bit 1 of DD01/9110, the Request to Send signal output.
9. Wait for the Data Terminal Ready signal to indicate the data terminal is ready by testing bit 2 of DD01/9110 in a loop until this bit becomes 1.
10. 64: LDA \$90. CLC. JMP EF3B to STA DD0D, EOR 02A1, ORA \$80, STA 02A1, STA DD0D, and RTS. Thus, FLAG interrupts are enabled, and 02A1 shows that the only active RS-232 interrupt is FLAG.

VIC: LDA \$90, STA 911E to enable CB1 interrupts, then CLC and RTS.

### **CHRIN from RS-232 Device** **F1B8/F26F-F1C9/F279**

#### **Called by:**

BEQ at F177/F22E in Determine Input Device.

JSR to the GETIN from RS-232 Device routine at F14E/F205, which returns with the accumulator containing either a character from the receive buffer or 0 if the receive buffer is empty. If the receive buffer is empty, it loops until a character other than 0 is returned. An infinite loop is possible on the VIC if the RS-232 receive buffer does not receive any more characters from the Received Data line. On the 64, however, the infinite loop should not occur since a return with \$0D in the accumulator occurs if the RS-232 status register indicates the Data Set Ready signal is missing.

#### **Operation:**

1. JSR F14E/F205 to get a character from the RS-232 receive buffer, with 0 returned if the buffer is empty.
2. If the carry is set, branch to step 5. However, this branch should never be taken, since the carry is cleared at exit from the subroutine at F14E/F205.
3. 64: If the character returned was 0, also test bit 6 of 0297, the status register, to see if the Data Set Ready signal is missing. If the DSR missing bit is 1, LDA \$0D, CLC, and RTS. If the DSR missing bit is 0, branch to step 1. The 64 actually tests both bits 6 and 5 of 0297 with AND \$60. This works since bit 5 is unused, but the code should actually be AND \$40 to test only for the DSR missing signal.

VIC: If the character returned was 0, branch to step 1.

4. CLC.
5. RTS.

### **GETIN from RS-232 Device** **F14E/F205-F156/F20D**

#### **Called by:**

Falls through from F14C/F1F5 in GETIN Preparation, JSR at F1B8/F26F in CHRIN from RS-232 Device.

This routine performs the following with a JSR to F086/F14F to get a character from the RS-232 receive buffer:

- If the head of the receive buffer is equal to the tail of the receive buffer, indicating an empty receive buffer, return with 0 in the accumulator.
- If the head and tail are not equal, return the character at the head of the receive buffer in the accumulator and increment the head pointer.

If you want larger buffers than the 256-byte (one-page) buffers for transmit and receive, you can change the pointers to the base locations for transmit (F9) and receive (F7) to point to your buffer locations. For ease of coding, you should probably maintain your buffers as exact multiples of a page. You need either to write your own OPEN routine to do this, or set the pointers after OPEN has completed. Also, you need to keep track of when the head and tail pointers are incremented from \$FF to \$00 and thus cross a page boundary, and to keep pointers to the receive and transmit buffers that keep track of what page you are in for the head and tail of both.

### **Operation:**

1. Save the Y register in 97.
2. JSR F086/F14F to get a character from the RS-232 receive buffer, or to get 0 if the buffer is empty.
3. Restore Y register from 97.
4. CLC and RTS.

### **Get Character from RS-232 Receive Buffer F086/F14F-F0A3/F15F**

#### **Called by:**

JSR at F150/F207 in GETIN from RS-232 Device.

If the head of the receive buffer is equal to the tail of the receive buffer, indicating an empty receive buffer, return with 0 in the accumulator.

If the head and tail are not equal, return the character at the head of the receive buffer in the accumulator. Also, increment the head.

#### **Operation:**

1. LDY 029C, the head of the receive buffer.
2. CPY 029B, the tail of the receive buffer.
3. If the tail and head are equal, the buffer is empty, so RTS with zero in the accumulator. Also, on the 64 only, set bit 3 of 0297 to 1 to indicate an empty receive buffer.



4. If the tail and head are not equal, return the character from the receive buffer head location in the accumulator. Also, on the 64 only, set bit 3 of 0297 to 0 to indicate the receive buffer is not empty.
5. Increment the head of the receive buffer, 029C. When the head reaches the end of the receive buffer, indicated by \$FF, this increment resets the head to \$00, the start of the circular receive buffer.
6. RTS.

### **Receive RS-232 Bit: NMI Interrupt Driven EF59/F036-EF6D/F04A**

#### **Called by:**

JMP at FF04/JSR at FF26 from NMI Interrupt Handler when NMI interrupt occurs for timer B of CIA #2/timer 2 of VIA #1.

This routine is called whenever a timer B/timer 2 interrupt on CIA #2/VIA #1 occurs, indicating that the time required to receive a bit has passed and it is now time to sample PB0, the Received Data line. To indicate the start of transmission of a byte to the 64/VIC, a FLAG/CB1 interrupt occurs. This interrupt then enables timer B/timer 2 interrupts and initializes the timer B/timer 2 bit time value.

Once a start bit has been received (a start bit has a value of 0, as opposed to the idle state of 1), reception of the data can begin. If a start bit has not yet been received, branch to a routine that checks for the start bit; if it doesn't find a start bit, the routine then enables FLAG/CB1 interrupts and disables timer B/timer 2 interrupts.

If the start bit has been received, all following bits are considered data bits until the counter for the number of bits received, A8, is decremented to 0.

Each of these data bits, found as bit 0 of A7 (set from PB0 of DD01/9110 when a timer B/timer 2 interrupt occurs), is rotated right into bit 7 of AA, the data byte being built. Each data bit is also Exclusive ORed with AB, the parity bit indicator, so that an even number of ones received makes AB zero, while an odd number of ones received makes AB one.

Whenever the counter of bits received, A8, reaches 0, indicating all the data bits have been received, branch to put the byte just received into the receive buffer.

If the counter of bits received is decremented below 0, branch to check for stop bits.

## RS-232 I/O Routines

It appears that two variables used in this routine, A9 (the receiver flag to check for a start bit) and AB (the parity bit indicator), are not initialized by any RS-232 routines before this routine is called the first time. Of course, at system reset A9 and AB are set to 0. These two bytes are also modified in tape operations. The lack of initialization could cause problems. For example, if A9 is 0 the first time this routine is entered, it appears no check for a start bit will be made on the first byte of data received over RS-232 lines. Also, since AB is not initialized either before or after this routine, the last value left in it will affect the parity indicator on future bytes received. Thus, individual byte parity checks may be incorrect.

It seems that A9 should have been initialized to a value other than 0 during the Open RS-232 Channel for Input routine. AB should have been initialized to the correct parity setting during the Open Logical File for RS-232 Device routine, and also reset to this value whenever a parity error occurs. This is corrected in version 3 of the 64 Kernal ROM: AB is set to \$01 after each start bit has been received.

### Entry requirements:

A9, check flag for start bit, should hold 0 if the start bit has been received, and should hold \$90 if the start bit has not been received.

AB, parity bit indicator, should hold 0 if an even number of 1's have been received so far or 1 if an odd number of 1's have been received so far.

A7, receiver bit temporary storage, should have bit 0 set to the bit received at PB0 when a timer B/timer 2 interrupt occurs.

AA, the byte being built, should contain the bits received so far, with the bits shifted into AA from high to low direction.

### Operation:

1. If the check for start bit flag, A9, is nonzero, branch to EF90/F068 to check for a start bit.
2. If the start bit flag is zero, indicating a start bit has been received, decrement the counter for the number of bits received, A8. This counter is originally set to 1 more than the number of bits in a word. Thus, it reaches 0 when all bits of data in the word have been received.
3. If A8 = 0, all bits of data have been received. Branch to EF97/F06F to store the byte received into the receive buffer.

4. If  $A8 < 0$ , branch to EF70/F04D to check for stop bits.
5. Exclusive OR the bit received at PB0, now in bit 0 of A7, with the parity bit indicator, AB, storing the result back into AB. An even number of 1's received makes  $AB = 0$ , while an odd number received makes  $AB = 1$ .
6. LSR the bit received at PB0, stored in bit 0 of A7 by the NMI Interrupt Handler routine, into the carry flag.
7. ROR the byte being built, AA. This shifts the bit received into bit 7 of AA.
8. RTS.

## Store Byte Received into Buffer EF97/F06F-EFB2/F08A

### Called by:

BEQ at E5F5/F03C in Receive RS-232 Bit.

If there is space left in the receive buffer, store the byte just received at the tail of the buffer. This byte has its unused high bits filled with 0's if its word length is less than eight.

The routine then falls through to EFB3/F08B to check the parity for the byte just received.

If the receive buffer is full, the character just received is discarded. Branch to EFCA/F0A2 to set the bit in the status byte, 0297, that indicates the receive buffer is full and prepare to receive the next byte.

With a circular queue such as the RS-232 receive buffer on the 64/VIC, the maximum number of characters the buffer can hold is equal to the size of the buffer minus one byte, which for the 64/VIC is 255 (decimal). If the tail plus one equals the head, the character cannot be stored in the last free byte (the tail location). If it was, the tail pointer would be incremented and would equal the head pointer—but this condition is used to determine when the buffer is empty. Thus, you could not distinguish the empty buffer condition from the full buffer condition. This same approach is also used for the RS-232 transmit buffer. If the buffer becomes full, the RS-232 status register is set to indicate it. However, no Kernal routines check for this condition, thus allowing you to continue to overrun the receive buffer.

### Operation:

1. If the tail plus one of the receive buffer (029B is the tail) is equal to the head of the receive buffer, 029C, the buffer is

- full, so branch to EFCA/F0A2 to set the bit in the status register, 0297, that indicates a receive buffer overrun error has occurred and prepare to receive the next byte.
2. If the buffer is not full, increment 029B, the tail of the buffer.
  3. LDA with the byte just received, AA, which still has the bits it received left justified.
  4. Check 0298 to see if the character length being received is less than eight. If so, shift in the correct number of high 0's to right justify the bits in this byte.
  5. Store the right justified received data byte that is now in the accumulator into the next available position in the receive buffer with STA (F7),Y. The two-byte pointer at (F7) points to the base location of the receive buffer and the Y register points to the location of the tail before the tail was incremented in step 2.
  6. Fall through to EFB3/F08B to check the parity for this byte.

### **Check Parity of Received Byte** **EFB3/F08B-EFC6/F09E**

#### **Called by:**

Falls through from EFB1/F089 Store Byte Received into Buffer.

Once the data byte has been received and stored in the receive buffer, this routine determines if any parity errors occurred. The RS-232 control register, 0294, specifies the type of parity.

If the parity is not being used, this routine branches to EF6E/F04B to check for stop bits (rather than checking for a parity bit).

If mark parity or space parity is being used, just RTS, as no parity check is performed on bytes received. However, the bit just received was a parity bit, so in effect throw this parity bit away before checking for stop bits.

If none of the above parity conditions are in effect, either odd or even parity must be in use. The parity bit indicator, AB (set to 0 if an even number of 1's are transmitted as data, or set to 1 if an odd number of 1's are transmitted), is Exclusive OR'd with the parity bit just received—bit 0 of A7. If the result of this Exclusive OR is 0 (the total number of 1's received including the parity bit is even), then odd parity is assumed and a parity error exists if even parity was used. Similarly, if

the result of the Exclusive OR is 1 (the total number of 1's received including the parity bit is odd), then even parity is assumed and a parity error exists if odd parity was being used. The 64/VIC implementation of odd and even parity is reversed from the accepted standard for parity. For even parity, the total number of bits with value 1, including the parity bit, should be even. For odd parity, the total number of bits with value 1, including the parity bit, should be odd. If you are trying to receive bytes from the RS-232 port using a parity check and the transmitting device is using the correct standard definitions of parity, you should set the command register to odd parity if the transmitting device is using even parity or vice versa. However, this problem has been corrected in version 3 of the 64 Kernal ROMs. The solution was to set AB, the parity bit indicator, to \$01 after each reception of a start bit for a byte, so that the final Exclusive OR with A7 gives the correct result.

## Operation:

1. Test the RS-232 control register, 0294, to see what kind of parity is being used.
2. If no parity is being used, branch to EF6E/F04B to check for a stop bit, rather than considering the bit just received as a parity bit.
3. If mark or space parity is being used, just RTS; the parity bit is not checked for received bytes.
4. If none of the above parity settings is in effect, Exclusive OR the parity bit just received, found now in bit 0 of A7, with the parity bit indicator, AB, that contains the result of the number of 1's transmitted as data. AB is 0 if an even number of 1's have been transmitted as data or 1 if an odd number of 1's have been transmitted as data.

If the result of this Exclusive OR is 0 (total number of 1's including parity is even), branch to step 6.

5. See what kind of parity is being used. If it's even, RTS by BVS EF6D/F04A. If odd, branch to step 7.
6. See what kind of parity is being used. If it's odd, RTS by BVC EF6D/F04A. If it's even, fall through to step 7.
7. Fall through to EFC7/F09F, the Handle Errors While Receiving from RS-232 Device routine, with a parity error indicated.

## Handle Errors While Receiving from RS-232 Device EFC7/F09F-EFDA/F0B2

### Called by:

Fall through from EFC6/F09E, Check Parity of Received Byte; alternate entry at EFCA/F0A2 by BEQ at EF9E/F076 in Store Byte Received into Buffer; alternate entry at EFCD/F0A5 by BEQ at EFDF/F0B7 in Check for Framing/Break Error; alternate entry at EFD0/F0A8 by BNE at EFDD/F0B5 in Check for Framing/Break Error.

Whenever an error occurs in the reception of RS-232 data, this routine is called to set the RS-232 status register, 0297, to indicate the cause of the error. The possible causes are: parity error, receive buffer overrun error, break detected, and framing error. The status register is set by an ORA of 0297. Thus, the status register is cumulative; more than one error condition can be recorded.

After setting the status register, jump to the routine to prepare for reception of the next byte.

### Operation:

1. LDA \$01 (parity error). Fall through to step 5.
2. EFCA/F0A2: LDA \$04 (receive buffer overrun). Fall through to step 5.
3. EFCD/F0A5: LDA \$80 (break detected). Fall through to step 5.
4. EFD0/F0A8: LDA \$02 (framing error). Fall through to step 5.
5. ORA 0297, the RS-232 status register, and store the result back into 0297.
6. JMP EF7E/F05B to prepare to receive the next byte.

## Check for Stop Bits EF6E/F04B-EF7D/F05A

### Called by:

BEQ at EFB8/F090 in Check Parity of Received Byte; alternate entry at EF7D/F04D by BMI at EF61/F0E3 in Receive RS-232 Bit.

When this routine is called, a stop bit should have been received. A stop bit is represented by a 1. Thus, if the bit just received was a 0, an error has occurred. This error is either a break detected error or a frame error, depending on whether the data byte received was 0 or not.

If a stop bit has been received, then this routine deter-

mines how many stop bits were specified in the RS-232 control register and whether all the stop bits have been received. If not all have yet been received, then exit. If all the stop bits have been received, then fall through to EF7E/F05B to prepare to receive the next byte from RS-232 transmission.

To determine whether all stop bits have been received, the stop bit indicator in bit 7 of 0293 is shifted into the carry; then the accumulator is loaded with 1 and an ADC to A8 is performed. This results in 0 if all stop bits have been received. An illustration of how this works follows:

*First Pass:*

Number of stop bits = 2

Carry = 1

Accumulator = 1

A8 = -1

Total = + 1 (so all stop bits have not yet been received)

*Second Pass:*

Number of stop bits = 2

Carry = 1

Accumulator = 1

A8 = -2

Total = 0 (so all stop bits have been received)

Similar calculations will show that this procedure correctly calculates stop bits for one stop bit and when no parity is being used.

**Entry requirements:**

Bit 7 of 0293, RS-232 control register, should contain 0 if one stop bit is being used, and should contain 1 if two stop are being used.

A7, receiver bit temporary storage, should have bit 0 containing the last bit received on PB0.

A8, counter of the number of bits received, should hold any one of the following values:

At entry point ED70/F04D, it will equal -1 if all data bits have been received (the bit just received should be either the single stop bit if the one stop bit is being used, or the first of two stop bits if two stop bits are being used).

The counter will hold -2 if two stop bits are being used, and the bit just received should be the second stop bit.

At entry point EF6E/F04B when no parity is being used, the counter will hold a 0.

## Exit conditions:

A8, the receiver bit count, is decremented if entering at EF6E/F04B when no parity is in effect.

Z flag of status register is 0 (BNE condition) if all stop bits have not yet been received.

Z flag of status register is 1 (BEQ condition) if all stop bits have been received.

## Operation:

1. Decrement the receiver bit count, A8.
2. EF70/F04D: If the bit just received on PB0 was 0, a stop bit was not received, as stop bits always have the value of 1. Branch to EFDF/F0B3 to set either the break detected flag or the frame error flag if a stop bit was not received.
3. If a stop bit was received, see if it was the last stop bit expected. Do this by shifting the stop bit indicator in bit 7 of 0293, the RS-232 control register, into the carry, and then adding 1 and the current receiver bit count, A8.
4. If the result of the above operation was not 0, RTS.
5. If the result of the operation in step 3 was 0, fall through to EF7E/F05B to prepare to receive the next byte.

## Check for Framing/Break Error EFDB/F0B3-EFE0/F0B8

### Called by:

BEQ at EE72/F04F in Check for Stop Bit.

If the check for a stop bit did not find a stop bit, but rather found a 0, an error in reception has occurred. This error is either determined to be a frame error if the data bits received included some bits set to 1, or is considered a break detected error if the data bits received so far were all 0's.

Branch to the appropriate error routine for setting the RS-232 status register.

### Operation:

1. If the byte received, AA, is nonzero, assume data bits were transmitted and this byte is bad; a frame error has occurred. Branch to EFD0/F0AB to indicate a frame error and then prepare to receive the next byte.
2. If the byte received, AA, was zero, assume a break was being sent. Branch to EFCD/F0A5 to set the break detected flag and prepare to receive the next byte.



## Prepare to Receive Next Byte EF7E/F05B-EF8F/F067

### Called by:

Falls through from EF7D/F05A in Check for Stop Bits, , BNE at EF92/F06A in Check for Start Bit, JMP at EFD8/F0B0 in Handle Errors While Receiving from RS-232 Device.

This routine is called whenever the computer is preparing to receive the next byte from the RS-232 port. It is needed whenever the previous byte has been received successfully, when the previous byte has a receive error of any kind, or when the check for a start bit does not find one.

This procedure enables FLAG/CB1 interrupts and disables timer B/timer 2 interrupts on CIA#2/VIA#1. The receiver flag for checking for a start bit, A9, is set to \$90, indicating that RS-232 receive is to check for a start bit.

### Operation (64):

1. LDA \$90.
2. STA DD0D, CIA #2 interrupt control register, thus enabling the interrupt mask for FLAG, RS-232 received data.
3. ORA 02A1, STA 02A1 to turn on bits 7 and 4, indicating that the FLAG interrupt is enabled.
4. STA A9, setting the receiver flag for checking for a start bit.
5. LDA \$02. JMP EF3B to disable timer B interrupts from CIA #2 and to indicate in 02A1 that timer B interrupts are disabled.

### Operation (VIC):

1. Enable CB1 interrupts for VIA #1.
2. Set the receiver flag for checking for a start bit, A9, to \$90.
3. Disable VIA #1 timer 2 interrupts.

## Check for Start Bit EF90/F068-EF96/F06E

### Called by:

BNE at EF5B/F038 in Receive RS-232 Bit.

Called when the flag for checking for a start bit indicates there should be a start bit, this routine sees if a start bit has been received by checking the bit just received from PB0 (stored in bit 0 of A7 by the NMI Interrupt Handler routine). This bit will be 0 if a start bit has been received, as start bits are always represented as 0's.

If a start bit has been received, clear the flag to check for a start bit, A9, and exit.

If a start bit has not been received, branch to the routine at EF7E/F05B which prepares for the reception of the next byte, and which resets the flag to check for a start bit.

## Operation:

1. See if bit 0 of A7 indicates a start bit has been received, as shown by a value of 0.
2. If a start bit has not been received, branch to EF7E/F05B to prepare to receive the next byte and to reset the flag to again check for a start bit.
3. If a start bit has been received, set the flag to check for a start bit located at A9 to 0 and RTS. (For version 3 of the 64 Kernal ROM, instead JMP E4D3 to a patch area, where A9 is set to 0, AB is set to 1, and RTS.)

## Open RS-232 Channel for Output EFE1/F0BC-F013/F0EC

### Called by:

JMP at F26C/F325 in CHKOUT Execution; alternate entry at F00D by JSR at F459 in Open Logical File for R2-232 Device and BPL at F05E in Open RS-232 Channel for Input (64); alternate entry at F0E8 by BPL at F127 Open RS-232 Channel for Input (VIC).

To prepare for RS-232 output, this routine must be executed. The output device number, 9A, is set to 2.

If 3-line handshaking is being used, CLC and RTS.

If x-line handshaking is being used, perform a handshaking sequence between the 64/VIC (the data terminal) and the the data communications equipment (DCE: modem, etc.). This handshaking sequence is: Data Set Ready must be on, Request to Send must be on, wait for any RS-232 input to complete, wait for Clear to Send to turn off, turn on Request to Send, test for Clear to Send to turn on. If Clear to Send is not on yet, continue to check for it as long as the Data Set Ready is still on. If this handshake sequence fails, set RS-232 status register to Data Set Ready missing.

The EIA RS-232 standard states that when the Clear to Send, Request to Send, and Data Set Ready conditions are on, any signals on circuit BA, the Transmitted Data line (CIA #2 PA2/VIA #1 CB2), will be transmitted to the communication

channel. The handshaking sequence for the x-line mode sees that these conditions are met.

## Operation:

1. Set the output device number, 9A, to 2.
2. If 3-line handshaking is being used, branch to step 13.
3. If x-line handshaking is being used, continue with step 4.
4. If DD01/9110 indicates Data Set Ready (from the DCE) is missing, branch to step 12. If Data Set Ready is on, continue with step 5.
5. If DD01/9110 indicates Request to Send is on (from the 64/VIC), branch to step 13. If Request to Send is off, continue with step 6.
6. 64: If 02A1 indicates timer B interrupts for RS-232 reception are enabled, loop until timer B interrupts are disabled.  
VIC: If 911E indicates interrupts are enabled for either timer 2 interrupts on VIA #1 or CB1 interrupts on VIA #1, loop until these interrupts are both disabled. This loop allows any RS-232 input to complete.
7. Test DD01/9110 for Clear to Send and loop until it is turned off by the data communications equipment.
8. Turn on Request to Send in DD01/9110.
9. If Clear to Send is on, branch to step 13. If it is off, go to step 10.
10. If Data Set Ready is still on, branch to step 9 to loop for Clear to Send to turn on.
11. If Data Set Ready is off, fall through to step 12.
12. F00D (64): store \$40 in 0297 to indicate Data Set Ready missing.  
F0E8 (VIC): JSR F016 to set Data Set Ready missing in 0297, and to enable timer 1 interrupts on VIA #1.
13. CLC and RTS.

## CHROUT to RS-232 Device F208/F2B9-F20D/F2C6

### Called by:

BCC at F1E3/BEQ at F28D in Determine Output Device.

A call to the Kernal CHROUT vector is routed here if the output device is RS-232. The routine retrieves the character to be output from 9E (64) or from the top of the stack (VIC). It then calls the routine to put this character in the transmit buffer and start transmission if this is the first byte to be sent.

## Entry requirements:

64: 9E should hold the character to be output; the X and Y registers should be saved on stack.

VIC: top of stack should hold the character to be output.

## Operation (64):

1. JSR F017 to put the character in the transmit buffer and to start transmission if this is the first byte to be sent.
2. JMP F1FC to restore the Y and X registers from the stack and restore the accumulator from 9E, the character that was output, then exit.

## Operation (VIC):

1. Pull the character to be output from the stack.
2. Save X and Y registers in 97 and 9E respectively.
3. JSR F0ED to store the character in the transmit buffer, and to enable timer 1 interrupts and set the timer 1 value if timer 1 interrupts are off.
4. Restore the X and Y registers from 97 and 9E.
5. CLC and RTS.

## Store Character in Transmit Buffer

**F014/F0ED-F02D/F101**

### Called by:

JSR at F208/F2BE CHROUT to RS-232 Device. (The actual entry point for 64 is F017.)

If the tail plus one of the transmit buffer is not equal to the head of the transmit buffer, continue. However, the transmit buffer is full if the head and tail plus one are equal, in which case wait until they are no longer equal by having 029D, the head of the RS-232 transmit buffer, incremented whenever a byte is sent. The character to be transmitted is then stored in the tail location of the transmit buffer.

Next, if timer A/timer 1 interrupts are already enabled, which indicates that RS-232 transmission is already in progress, simply RTS. However, if timer A/timer 1 interrupts are not enabled, fall through to F02E/branch to F102 to set timer A/timer 1, enable timer A/timer 1 interrupts, and send the new byte.

While the receive buffer loses characters coming in when it is full, the transmit buffer just waits for a character to be transmitted and space to open up in the buffer; it does not lose characters if the buffer is full.

## Operation:

1. F014 (64): JSR F028, step 7, to see if timer A interrupts are enabled. If not, load the timer A bit time value from 0299–029A, enable timer A interrupts, and send the new byte.
  2. F017/F0ED: LDY with the tail of the transmit buffer, 029E.
  3. INY.
  4. If the Y register is now equal to the head of the transmit buffer, 029D, wait until the head of the transmit buffer is incremented by looping to step 2. On the 64, however, loop to F014 (step 1), which then falls through to step 2.
  5. STY in 029E, the tail of the transmit buffer.
  6. Store the character to be transmitted at the location of the tail of the transmit buffer before the tail was incremented. That is, DEY and store character at (F9),Y.
  7. F028 (64): If timer A interrupts are enabled, RTS. If timer A interrupts are not enabled, fall through to F02E to set timer A, enable timer A interrupts, and send new byte.
- F0FC (VIC): If timer 1 interrupts are not enabled, branch to F102 to set timer 1, enable timer 1 interrupts, and send new byte. If timer 1 interrupts are enabled, RTS.

## Prepare Timer A/Timer 1 Interrupt for RS-232

### Transmission

#### F02E/F102–F04C/F115

### Called by:

Fall through from F02D/BVC at F0FF in Store Character in Transmit Buffer.

No timer A/timer 1 interrupts for CIA #2/VIA #1 are enabled when this routine is called, indicating that RS-232 transmission is not taking place. This routine prepares for RS-232 transmission by setting timer A/timer 1 to the value for one bit time and then enabling timer A/timer 1 interrupts for CIA #2/VIA #1. Finally, jump to the routine to prepare for transmission of the next byte.

## Operation:

1. Set timer A (DD04–DD05) of CIA #2/timer 1 (9114–9115) of VIA #1 to the value in 0299–029A, which contains the time for transmission of one bit.
2. 64: LDA \$81 and JSR EF3B to enable the interrupt mask for timer B interrupts in the CIA #2 interrupt enable register and to indicate in 02A1 that timer B interrupts are enabled.

VIC: Set 911E, VIA #1 interrupt enable register, to enable timer 1 interrupts.

3. JSR EF06/JMP EFEE to the routine to prepare to transmit the next byte.
4. 64: Store \$11 in DD0E to load timer A counter from its latches and to start the timer, then RTS.

## Prepare to Transmit Next Byte EF06/EFEE-EF2D/F015

### Called by:

JSR/JMP at F044/F113 in Prepare Timer A/Timer 1 Interrupt for RS-232 Transmission, BEQ at EEBD/EFA5 in Transmit RS-232 Bit: NMI Interrupt Driven.

If 3-line handshaking is being used, no check of any handshake sequence is done.

If x-line handshaking is being used, an attempt is made to see if the Data Set Ready and Clear To Send lines are still on. However, again the VIC-20 refers to the wrong location, using 9120 rather than 9110, thus not accurately testing these conditions. The 64 does a correct check of DD01 for these conditions.

The next bit to be sent, the B5 variable, is set to 0, as is the parity indicator, BD. The number of bits to be transmitted is stored in B4.

Next, test to see if the head of the transmit buffer equals the tail of the transmit buffer, indicating the buffer is empty. If the buffer is empty, timer A/timer 1 interrupts are disabled since there is nothing to send.

If the transmit buffer is not empty, get the next byte to be transmitted from the RS-232 transmit buffer and store this byte in B6, which is the byte from which bits will be picked off and transmitted. Finally, increment the head of the transmit buffer.

### Operation:

1. See if 3-line handshaking is being used, and if so branch to step 3.
2. 64: For x-line handshaking, test DD01 to see if the Clear To Send and Data Set Ready lines are still on. If so, continue with step 3. If not, branch to EF31 to set the bit in the status register that indicates Clear To Send is missing, or to

EF2E to set the status register bit that indicates Data Set Ready is missing.

VIC: For x-line handshaking, test 9120 bits 7 and 6 in an incorrect attempt to check for Data Set Ready and Clear To Send on. Location 9110 should have been used, as noted throughout this chapter. Normally, though, 9120 bits 7 and 6 are always 1 as they are used in scanning the keyboard for the STOP key. Thus, x-line handshaking on the VIC will never detect either DSR or CTS missing. Continue with step 3 if both bits are 1.

3. Store 0 in BD, the parity indicator, and in B5, the next bit to be transmitted.
4. Set B4, the number of bits plus one to transmit, from the value in 0298.
5. See if the head of the transmit buffer, 029D, is equal to the tail of the transmit buffer, 029E. If equal, the buffer is empty, so branch to EF39/F021 to disable timer A/timer 1 interrupts for RS-232 transmission.
6. If the head and tail are not equal, retrieve the next byte from the transmit buffer pointed to by the head of the transmit buffer and save this byte in B6, the byte to be transmitted.
7. Increment the head of the transmit buffer, 029D.

## Handle Errors While Transmitting to RS-232 Device EF2E/F016-EF49/F026

### Called by:

BPL at EF0F/FFF7 in Prepare to Transmit Next Byte, JSR at F0E8 in Open RS-232 Channel for Output (VIC), JMP at F518 in Open Logical File for RS-232 Device (VIC); alternate entry at EF31/F019 by BVC at EF11/FFF9 in Prepare to Transmit Next Byte; alternate entry at EF39/F021 by BEQ at EF24/F00C in Prepare to Transmit Next Byte; alternate entry at EF3B JMP at EF8D in Prepare to Receive Next Byte, JSR at F041 in Prepare Timer A/Timer 1 Interrupt for RS-232 Transmission, and JMP at F07A in Open RS-232 Channel for Input (64).

The RS-232 status register, 0297, is set to Data Set Ready missing or to Clear To Send missing, and then timer A/timer 1 interrupts are disabled.

It appears that the entry at F019 for the VIC never occurs due to the now familiar incorrect test of 9120 for Clear To

Send. Also, at F016 on the VIC, the only entry possible of the three listed is from F0E8 since the other two entries also check 9120 rather than 9110. Entry from F0E8 is made if both Clear To Send and Data Set Ready are missing. However, only Data Set Ready missing is set. Thus, Clear To Send missing is never set on the VIC.

## Operation:

1. Prepare to turn on bit 6 of the status register (to indicate that the Data Set Ready signal is missing). Branch to step 3.
2. EF31/F019: Prepare to turn on bit 4 of the status register (to indicate that the Clear To Send signal is missing). Fall through to step 3.
3. Set status register, 0297, as indicated from step 1 or step 2.
4. F021 (VIC): Disable timer 1 interrupts on VIA #1 for RS-232 transmit.  
     EF39 (64): LDA \$01 to prepare to disable timer A interrupts for CIA #2.
5. EF3B (64): STA DD0D, EOR 02A1, STA 02A1, STA DD0D. The possible values in the accumulator at entry to step 5 are: if falling through from step 4, \$01 to disable timer A interrupts; if coming from a jump at EF8D, \$02 to disable timer B interrupts; if jumping from F07A, \$90 to enable FLAG1 interrupts; if jumping from JSR at F041, \$81 to enable timer A interrupts. The flag for interrupts enabled in 02A1 is also set.

## Transmit RS-232 Bit: NMI Interrupt Driven EEBB/EFA3-EED6/EFBE

### Called by:

JSR at FE9D/FEFC in NMI Interrupt Handler; alternate entry at EED1/EFB9 by BPL at EEEF/EFD4 and BNE at EEF0/EFD8 in Prepare Parity Bit and Set Counter for Stop Bits.

Whenever an NMI interrupt occurs the NMI interrupt handler checks to see what caused the interrupt. On the 64, if a timer A interrupt from CIA #2 occurs, this routine is called. On the VIC, if a timer 1 interrupt from VIA #1 occurs, this routine is called.

This routine prepares the next bit to be transmitted. This next bit will be sent when the next timer A/timer 1 interrupt occurs and is serviced by the NMI Interrupt Handler routine. The bit is placed in bit 2/bit 5 of B5.



First, see if the count of transmitted bits for this byte has reached 0 or is less than 0. If the count is 0, branch to retrieve the next byte from the transmit buffer and prepare it for transmission. If the count is less than 0, branch to send a stop bit.

If the count of bits to transmit is not yet 0, the byte being transmitted, B6, is shifted right one bit into the carry. If the carry is set by this shift, the X register is loaded with \$FF; otherwise, X is \$00.

If the bit shifted into the carry was a 1, EOR BD, the parity work byte, with either \$FF or \$00. Then decrement the transmit bit counter, B4. If the counter is now 0, branch to calculate the parity. Since the counter is originally set to one more than the number of bits in a word, it reaches 0 only after all bits in the word have been transmitted.

Finally, transfer the X register to the accumulator, with \$FF indicating the bit shifted from B6 into the carry was a 1, while \$00 indicates it was 0. The accumulator is then ANDed with \$04/\$20 and the result stored in B5, the next bit to be sent when the next timer A/timer 1 interrupt occurs. ANDing with \$04/\$20 sets bit 2/5 of B5. Later in the VIC NMI interrupt handler for timer 1 interrupts, this bit is ORed with 110 in bits 7-5 to produce the CB2 line control. Thus if bit 5 is 1, CB2 is held high at the logic 1 level, while if bit 5 is 0, CB2 is held low at the logic 0 level. The NMI interrupt handler for the 64 first ANDs the accumulator with \$FB (1111 1011 binary) to clear bit 2, then ORA B5 to set bit 2 to 1 or 0 depending on what bit value is to be sent. Then the accumulator is stored in DD00, thus placing this bit value onto CIA #2 data port A bit 2, which is the RS-232 data output line.

### Operation:

1. If B4, the transmit bit count, is 0, branch to EF06/EFEE to prepare to transmit the next byte from the transmit buffer.
2. If B4 < 0, branch to EF00/EFE8 to send a stop bit.
3. Shift the byte being transmitted, B6, right one bit into the carry flag (with a 0 shifted into bit 7 of B6).
4. Set X register to \$00.
5. If the shift in step 3 cleared the carry, branch to step 7.
6. Set X register to \$FF if carry was set by shift in step 3.
7. TXA and then Exclusive OR this value with BD. Thus, if the bit shifted into the carry in step 3 was a 1, this Exclusive OR will flip all bits in BD, while if the bit shifted was

- a 0, BD is left unchanged. BD is used to indicate whether an even or odd number of bits have been sent as 1's.
8. Decrement the transmit bit counter, B4, and if it is now 0, branch to EED7/EFBF to calculate the parity, as all the data bits have now been transmitted.
  9. EED1/EFB9: TXA again, with \$FF representing a 1 shifted into the carry bit in step 3, and \$00 a 0 shifted into the carry.
  10. AND with \$04/\$20 (bit 2/5 is the only bit possibly turned on) and store in B5, the next bit to be transmitted.
  11. RTS.

### **Prepare to Send Stop Bit EF00/EFEB-EF05/EFED**

#### **Called by:**

BMI at EEBF/EFA7 in Transmit RS-232 Bit.

A stop bit, which is a data value of 1, is prepared to be sent as the next bit to be transmitted. The correct number of stop bits is maintained by the transmit bit counter being set to a value of  $-2$  or  $-1$  in the routine to prepare the parity bit and set counter for stop bit (or  $-1$  or 0 if no parity was being used). The counter is incremented in this routine. Whenever the timer A/timer 1 interrupt service routine for transmitting RS-232 bits finds B4 less than 0, branch here to send a stop bit. This routine is executed once if one stop bit is being sent or twice if two stop bits are being sent (or just once for two stop bits if no parity is in effect.)

#### **Entry requirements:**

B4, transmit bit counter, should hold  $-1$  if one more stop bit is to be transmitted, and should hold  $-2$  if two more stop bits are to be transmitted.

#### **Exit conditions:**

B4 holds 0 if this is the final stop bit to be transmitted, and holds  $-1$  if one more stop bit is to be transmitted. The X register holds \$FF to prepare to send a stop bit value of 1.

#### **Operation:**

1. Increment B4.
2. LDX \$FF to prepare for a stop bit value of 1.
3. Branch to EED1/EFB9 (step 9 of the Transmit RS-232 Bit routine) to set the next bit to be transmitted.

## Prepare Parity Bit and Set Counter for Stop Bits EED7/EFBF-EEF1/EFE7

### Called by:

BEQ at EECF/EFB7 in Transmit RS-232 Bit.

This routine is called once all the data bits in a word have been sent. It determines the type of parity in effect—no parity, even parity, odd parity, mark parity, or space parity—and calculates a value for the parity bit in accordance with the type of parity being used. It also prepares a counter for the number of stop bits to be sent to allow the interrupt service routine to know how many stop bits to transmit.

If no parity is being used, the data line is set to 1, its idle state, and the number of stop bits is calculated.

If even or odd parity is being used, the parity work byte, BD, is tested to see if it is 0, indicating an even number of data bits have been transmitted with the value of 1. If odd parity is in effect, the parity bit is set to cause an odd number of 1's (including the parity bit) to be sent. Therefore, if odd parity is used and the parity work byte shows that an even number of bits have been sent with a value of 1, the parity bit transmitted will be a 1 to add up to an odd number of parity bits sent with a value of 1. Even parity works the same except that an even number of bits are sent with the value of 1. Notice that the 64/VIC uses the normal definition of even and odd parity when transmitting, in contrast to their use when receiving.

If mark parity is in effect, a parity bit of 1 is transmitted.

If space parity is in effect, a parity bit of 0 is transmitted.

Also, this routine checks the number of stop bits specified in the RS-232 control register and sets the transmit bit counter, B4, such that the correct number of stop bits will be sent.

### Exit conditions:

The X register contains \$FF if a parity bit of 1 is to be transmitted, and \$00 for a parity bit of 0.

B4, the transmit bit counter, is -1 if one stop bit is to be transmitted, -2 if two stop bits are to be transmitted, 0 if one stop bit is used and parity is not generated, and -1 if two stop bits are used without parity.

### Operation:

1. Test the parity setting of the RS-232 command register, 0294, bits 7-5.

2. If parity is not being used, branch to step 11.
3. If the parity check is disabled, branch to step 13 (for mark or space parity).
4. If even parity is being used, branch to step 12.
5. For odd parity if the parity indicator work byte, BD, is not 0 (indicating an odd number of bits were transmitted as 1's), branch to step 7.
6. Decrement the X register, thus the X register is now \$FF, indicating a parity bit of 1 will be sent.
7. Decrement the transmit bit counter, B4. It should now be -1 (unless parity is not being used, in which case B4 should equal 0 now).
8. See how many stop bits are specified in the RS-232 control register. If one stop bit is used, branch to EED1/EFB9, a point in the interrupt service routine (Transmit RS-232 Bit) that prepares the next bit to be transmitted from the current value of the X register.
9. If two stop bits are to be transmitted, decrement B4, the transmit bit counter. B4 should now be -2 (or -1 if no parity is being used).
10. Branch to EED1/EFB9 (see step 8).
11. If parity is not being used, increment the transmit bit counter, B4, so that it is now 1. This setting results in no parity bit being sent; rather, a stop bit is sent (or two if specified in the control register). Then branch to step 6 to prepare to send a 1 (the value for a stop bit).
12. If even parity is being used, see if the parity indicator work byte, BD, is 0. If so, branch to step 7 (to send a 0 as a parity bit) as an even number of 1's have already been sent. If not 0, branch to step 6 to send a 1 as an odd number of 1's have already been sent.
13. If space parity is being used, branch to step 7 to transmit a 0 as a parity bit.
14. If mark parity is being used, branch to step 6 to send a 1 as a parity bit.

## Close Logical File for RS-232 Device F2AB/F364-F2C7/F38C

### Called by:

Fall through from F2AA/F363 in Determine Device for CLOSE.

Whenever a CLOSE for an RS-232 file is performed, this routine is executed.

First, call another routine, which decrements the number of open files and removes the entry for this file from the logical file table, device number table, and secondary address table.

Next, disable interrupts for VIA #1 for all conditions except the RESTORE interrupt (VIC) or disable interrupts for all conditions on CIA #2 (64).

On the VIC, the Request to Send and Data Terminal Ready conditions are set on. Then the VIC also holds the transmitted data line high, which is the idle state for RS-232 I/O.

Finally, reset the pointers to the end of memory to include these two 256-byte buffers.

A couple of notes about CLOSE: Both *VIC Revealed* and the *VIC Programmer's Reference Guide* recommend that you check that all data is transmitted before doing a CLOSE; both references also recommend that you check bit 6 of 911F for a value of 1, indicating that data is still being transmitted. While these recommendations may be valid, bit 6 has no documented use for RS-232 I/O in the VIC schematic. The schematic does show that this bit is connected to pin 8 of the user port, but pin 8's use is not defined. The *Commodore 64 Programmer's Reference Guide* recommends checking ST for a value of 0 or 8 (the latter indicates an empty receive buffer) before closing the RS-232 device.

## Entry requirements:

The top of the stack should contain an index into the logical file number table for this file.

## Operation:

1. Pull the index into the logical file number table for this file from the stack.
2. JSR F2F2/F3B2 to decrement the number of open files and to remove the entry for this file from the logical file number table, the device number table, and the secondary address table.
3. 64: JSR F483 to disable all interrupts from CIA #2 and store 0 in 02A1. Continue with step 6.

VIC: Set 911E, the VIA #1 interrupt enable register, to disable interrupts for all conditions except the RESTORE key interrupt.

4. VIC: Set 9110 for Data Terminal Ready and Request to Send on.
5. VIC: Set 911C to hold CB2, Transmitted Data, high.
6. JSR FE27/FE75 to LDX from the low byte of the pointer to the top of memory, and to LDY from the high byte of the pointer to the top of memory.
7. If the high byte of the pointer to the RS-232 receive buffer, F8, is not 0, an RS-232 receive buffer is still allocated. In this case, INY. The Y register holds the pointer to the high byte of the end of memory. This INY reclaims one page (256 bytes) of storage from the receive buffer.
8. If the high byte of the pointer to the RS-232 transmit buffer, FA, is not 0, an RS-232 transmit buffer is still allocated, in which case INY to reclaim that page (256 bytes) of storage from the transmit buffer. Thus, if both buffers were still active, two pages (512 bytes) have been reclaimed from the buffers for user memory.
9. Reset the high bytes of the pointers to the receive and transmit buffers to 0, indicating they are no longer allocated.
10. JMP F47D/F53C to reset the pointer to the end of memory (0283) from the current values of the X and Y registers. Also, exit with the carry set and the accumulator containing \$F0.

## Disable RS-232 During Serial or Tape I/O F0A4/F160-F0B6/F173

### Called by:

JSR at ED0E/EE19 in Send TALK or LISTEN Command to Serial Device, JSR at F88A/F8FC in Common Tape Read and Write.

If RS-232 interrupts are enabled, wait for the RS-232 operation that set these interrupts to finish, then disable all RS-232 interrupts. Thus, this routine prevents RS-232 operations from interfering with the tape or serial operations which call this routine. It also makes sure that RS-232 operations have first priority and are completed before tape or serial I/O begins.

### Operation:

1. PHA.
2. LDA from 02A1/911E.

3. If the accumulator is 0, indicating no interrupt conditions for RS-232 are enabled, branch to step 10.
4. LDA from 02A1/911E.
5. 64: AND \$03 (binary 0000 0011). Bit 0 is the timer A interrupt enabled status, and bit 1 is the timer B interrupt enabled status.  
VIC: AND \$60 (binary 0110 0000). Bit 6 is the timer 1 interrupt enabled status, and bit 5 is the timer 2 interrupt enabled status.
6. BNE to step 4 until timer B/timer 2 and timer A/timer 1 interrupts are both disabled. Timer A/timer 1 interrupts for RS-232 transmit are disabled when the logical file is closed. Timer B/timer 2 interrupts for RS-232 receive are disabled after each byte is received.
7. LDA \$10.
8. STA DD0D/911E to disable FLAG/CB1 interrupts.
9. 64: Store 0 in 02A1 to indicate no RS-232 interrupts are enabled.
10. PLA.
11. RTS.





## **Chapter 10**

---

# **Tape I/O Routines**



# Tape I/O Routines

---

With the cassette tape unit on the 64/VIC you can store and retrieve sequential data. This data can be programs (BASIC and machine language) or the contents of contiguous areas of memory. The stored information can also be sequential files where the data has been sent character by character to a 192-byte buffer which is dumped to tape when full or when the file is closed.

Each file of information on tape is preceded by block of data called a *header*, which contains information about the file. A tape identifier is the first byte for a header. The identifier byte indicates whether the following block is sequential data, a program, or an end-of-tape marker. Sequential files start with a header that has an identifier byte of 4, followed by 192-byte blocks of saved data, each starting with an identifier byte of 2. Program files contain headers with identifier bytes of 1 or 3, followed by the program data (without any tape identifier). Both program and sequential files are optionally followed by an end-of-tape header with a tape identifier of 5. When storing data to tape, if you use BASIC's SAVE or the Kernal's SAVE routine, the stored data will be in program format. When retrieving data from tape, program data will normally be retrieved through BASIC's LOAD or the Kernal's LOAD routine. Data recorded in sequential format will usually be retrieved in through BASIC's OPEN, GET# or INPUT#, and CLOSE or the Kernal's OPEN, CHRIN, and CLOSE routines.

You can, however, treat the program data as a sequential file and use BASIC's OPEN, GET# or INPUT#, and CLOSE or the Kernal's OPEN, CHRIN, and CLOSE routines to read the data. Normally, you wouldn't try to retrieve data stored in sequential file format with BASIC's LOAD or the machine language LOAD routine.

When sequential files are used, the bytes that are sent to tape are first collected in the 192-byte cassette buffer at 033C; each time the buffer fills it is dumped to tape and when the file is closed the last partially filled buffer is dumped to tape.

## Tape I/O Routines

Data stored as a sequential file thus appears on tape as a ten-second leader followed by a 192-byte header that contains a tape identifier of 4, the starting and ending address of the the tape buffer, and the filename of the sequential file. (A leader is just the tape section preceding a block of data; the leader contains recorder pulses used in adjusting the software tape reading mechanism. The header is stored twice in two identical blocks—except for block countdown characters). Following this header are the sequential data areas—192-byte buffers that have been dumped to tape.

Each 192-byte buffer is stored as two identical blocks with a short interblock gap between them (in the same format as the header for the sequential or program files, with block countdown characters and a checksum for each of the two blocks that make up the buffer). Between one two-block 192-byte buffer and the next is another leader, however it is only about three seconds long. When the tape file is closed, the buffer is dumped to tape with its partially filled contents followed by a byte with a value of zero. A sequential file can also be followed by an end-of-tape header.

Data stored in program format is stored on tape in a different general format than sequential data, although the individual bytes are written in the same manner. A header is stored on the tape followed by the program. The header gives the starting and ending address of the following program. The header also contains as its first byte a tape identifier that indicates whether the program that follows is relocatable (tape identifier of 1) or non-relocatable (tape identifier of 3). A tape identifier of 5 is also used as an end-of-file header. This end-of-file header has exactly the same information as the one at the start of the program except for the identifier. It is used to prevent loads from going beyond this point on the tape. The header also contains the filename, if any, of the program that follows.

When doing a BASIC SAVE or using the Kernal SAVE routine, if you specify an even-numbered secondary address, the tape identifier which is placed on the tape is 1 (relocatable program file). If the secondary address is odd, the tape identifier is 3 (nonrelocatable program file). If the secondary address specified (which is contained in B9) has bit 1 set to 1 (for example, this would be true for a secondary address of 2 or 3), the end-of-file header with a tape identifier of 5 is written

following the program (or the sequential file). A relocatable program tape refers to one for which you may specify the starting address in the X and Y registers when you do a load. Just because you can relocate a program to a different area than it was saved in does not necessarily mean the program will execute correctly in this new area—if it is a machine language program, it must not have any absolute address references to the area from which it was saved, for example.

A nonrelocatable program tape is one that can't be relocated to a different starting area when you do a load, no matter what you specify in the X and Y registers or in the secondary address. Indeed, if you want to create a nonrelocatable program tape, save it with an odd secondary address, thus writing a tape identifier of 3. Thus you make certain that upon load the program will be loaded back to the correct location and that no secondary address is needed for the load. Even supposedly nonrelocatable tapes can be loaded back to a different area than the program was saved from, though. For example, you can specify your own load addresses for a program after loading the header, then jump to the portion of the tape load routine that loads the program. Similarly, if you understand how the tape routines work, you can do such things as save program data to tape without headers, create tapes that upon loading begin execution immediately, save to tape from memory beginning at or above location 8000 (on the VIC, saves above 8000 are normally prohibited), and various other unusual effects.

Figure 10-16 (page 318) illustrates the format of program tape storage. Both the header and the program are each recorded twice, as two identical blocks separated by an interblock gap. Leaders precede the various blocks on the tape; about a ten second leader precedes the first header block; a leader less than one second long appears between header blocks one and two and between program blocks one and two; and about a three second leader separates the header from the first program block.

In addition to the actual data stored in that block, each contains control characters. At the front of each block are nine block countdown characters. The countdown characters that precede block one have their high bits on; those before block two have their high bits off. The tape load routines examine the block countdown characters when they encounter a new

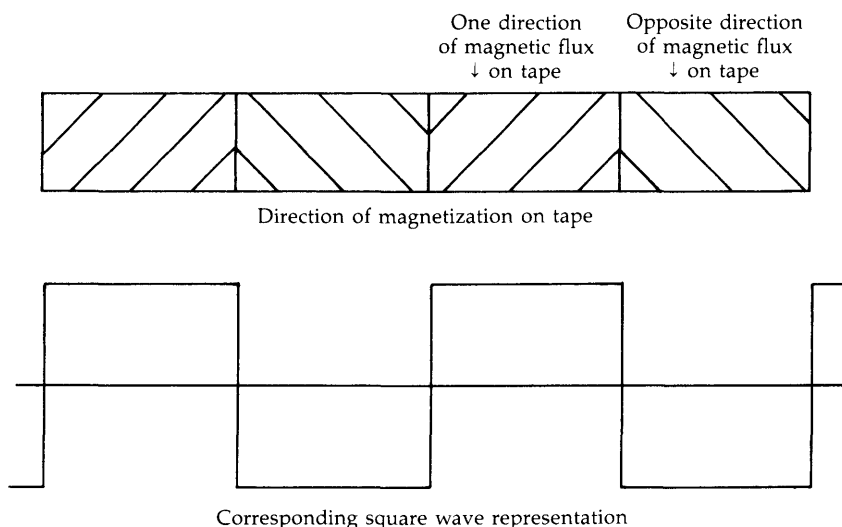
block to make sure that the block matches what the routines expect at that point. For example, if the tape routines have just completed reading program block one, the next block the routines expect is program block two, which would have block countdown characters with their high bits off. If the block countdown characters do not match what is expected at that particular point, the tape load routines keep searching through the tape to try to find the block countdown characters that match.

Another control character for each block is a checksum that is recorded as the last byte of each block. This checksum is the parity of all bytes that were saved for this block. When loading blocks, a checksum is also computed over all the bytes loaded. (This is done for both the header and the program.) The checksum for the header computed during the load is compared to the last byte of header block two, the header's checksum during the save. If not equal, a checksum error is flagged. Similarly, the checksum for the program computed during the load is compared to the last byte of program block two, the program's checksum during the save, and if they are not equal, a checksum error is indicated. It appears possible that a checksum error could occur on an otherwise perfect load where the loaded area is exactly the same as the saved area. If block two contained parity errors during its save operation, its checksum would differ from the error-free loaded combination of save blocks one and two. Also, while quite rare, double bit errors (an even number of bits in a byte are incorrect) occurring in bytes in both save block one and two could result in a load where the load checksum matched the save checksum but the loaded area is actually different than the saved area.

### Looking Closer

The above discussion talks about how tapes are recorded when you just view their format from the level of headers, blocks, interblock gaps, and leaders. Tape storage can be viewed on the bit level as well. When writing to tape the signal to the tape write line is reversed, which causes the polarity of the signal being written to tape to reverse. Figure 10-1 shows this model of tape magnetization.

## Figure 10-1. Model of Magnetic Tape Magnetization

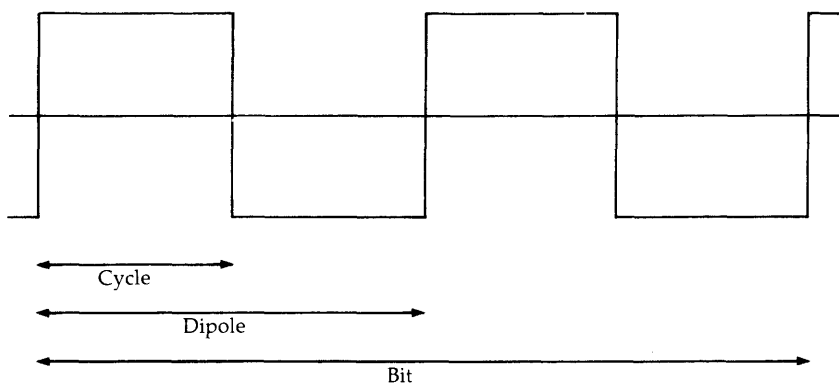


Each bit that is written to tape is broken up into four of these cycles, of original polarity/reversed polarity/original polarity/reversed polarity. Four of these cycles make a bit, whether that bit is a 0, a 1, a leader bit, or a word marker bit. The first two cycles make up the first dipole (literally, *two poles*), while the second two cycles make up the second dipole. To differentiate between leader bits, 0 bits, 1 bits, and word marker bits, the cycles are written for varying durations. For the particular U.S. NTSC VIC we have tested, leader (or leader-0) cycles were of approximately 174 microseconds, 0 cycles were of approximately 169 microseconds, 1 cycles were of approximately 247 microseconds, and word marker cycles were of approximately 332 or 328 microseconds. The way these times were derived is explained later. When writing the four cycles for a bit to tape, the cycles are broken up into two dipoles of two cycles each. For a leader bit the tape write routines put out all leader-0 cycles, or two leader dipoles. For a 0 data bit, the routines output two 0 cycles followed by two 1 cycles (a 0 dipole followed by a 1 dipole). For a 1 data bit, the routines put out a 1 dipole followed by a 0 dipole. For a word marker bit, a word marker dipole followed by a 1 dipole.

## Tape I/O Routines

While tapes are thus written in cycles, they are read in dipoles, where each negative (high-to-low) transition on the FLAG/CA1 tape read line frames a dipole. Figure 10-2 illustrates the meanings of the terms cycle, dipole, and bit as applied to tape.

**Figure 10-2. Cycle, Dipole, and Bit**



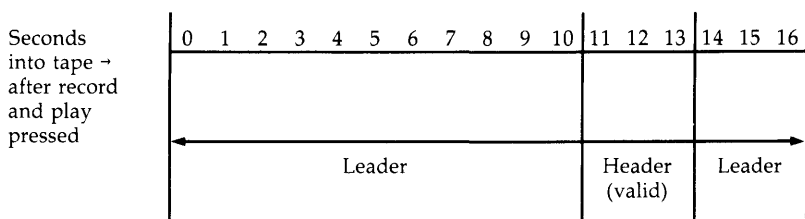
Various techniques allow for some variation in the speed at which tape is recorded and read back. The tape read routines are treated in much more extensive detail later in this chapter.

You may wonder why it is not normally possible to save to tape on the VIC at addresses greater than or equal to 8000 (32768 decimal). The reason is that during tape saves AC-AD is the pointer to the area being saved, and the high byte of the pointer (AD) is also used as an indication that a block has been completely saved. When all the bytes in a block have been saved and the checksum has also been saved, AD has its high bit turned on. The save routine checks AD to see if this high bit is on. When it detects the high bit on, the tape save then writes the leader between blocks 1 and 2, or turns off the tape motor if it just finished block 2. Having the high bit on in AD means that its value is at least \$80. Thus, a save with (AC) of greater than or equal to 8000 looks the same as the end of block flag, and when trying to save from such an area only the header is written, followed by the leader between the header and the expected program (which is not recorded) followed by the leader between blocks 1 and 2. Figure 10-3 illustrates a tape saved from greater than or equal to 8000 on the VIC.



## Tape I/O Routines

**Figure 10-3. Saving to Tape Above 8000 on the VIC**



This prevention of saving to tape from greater than or equal to 8000 is not carried over into the Commodore 64. The 64 will save to tape from above 8000 since it no longer uses AD for two conflicting purposes.

Here are a couple of questions you also might have about the tape routines:

*Can you record on both sides of a cassette tape?* Yes, separate tracks are used, as recording on the back side does not wipe out the programs on the front side.

On the subject of errors during tape loads, experiments with creating even just a one second dropout for program block one caused more than 31 errors and thus produced an unrecoverable read error. The maximum amount of time allowed for the total amount of dropouts should be 31 times the length of time to read a byte ( $832.4 \times 9$  [8 data bits + parity bit] + 1154.7 for word marker bit =  $7491.6 + 1154.7 = 8646.3$  microseconds per byte  $\times 31 = .268$  second maximum dropout).

*Can machine language programs be loaded from BASIC?* Yes. If you don't want your machine language programs to relocate upon loading by BASIC, you can guarantee they will be nonrelocatable by saving the machine language program with an odd secondary address to create a tape identifier of 3. However, if you are loading a machine language tape and are not sure what kind of tape identifier it has, you can also assure a nonrelocatable tape load by specifying a nonzero value for your secondary address, like `LOAD "TEST",1,1`.

### Program Examples

Following are examples of some ways in which to use the tape routines from machine language and from BASIC.

Figure 10-4 shows how you might do a machine language save; Figure 10-5 shows how to do a nonrelocatable machine language load; Figure 10-6 shows how to do a relocatable machine language load; Figure 10-7 shows how to write to sequential files using machine language; Figure 10-8 shows how to read from sequential files using machine language; Figure 10-9 shows how to save BASIC programs to tape; Figure 10-10 shows how to load a BASIC program from tape; Figure 10-11 shows how to verify a program from BASIC; Figure 10-12 shows how to write to a sequential file from BASIC; and Figure 10-13 shows how to read from a sequential file from BASIC.

### Figure 10-4. Machine Language Save

```
LDA $01 ; logical file number
LDX $01 ; device number for tape
LDY $01 ; odd secondary address for nonrelocatable file, even
          for relocatable
JSR FFBA ; SETLFS
LDA $09 ; number of characters in name
LDX $40 ; X and Y contain address of
LDY $04 ; filename, in this case at 0440
JSR FFBD ; SETNAM
LDX $FE
LDA $00 ; low address of start of save
STA $00,X ; zero page location (FE)
LDA $20 ; high address of start of load, in this case 2000
STA 01,X ; zero page location
TXA ; accumulator has pointer to zero page location of start
      of save
LDX $50 ; low byte of ending address + 1 of save area
LDY $25 ; high byte of ending address + 1 of save area, in this
          case 2550
JSR FFD8 ; SAVE
```

**Figure 10-5. Machine Language Load (Nonrelocatable)**

LDA \$01 ; logical file number  
LDX \$01 ; device number for tape  
LDY \$01 ; nonzero secondary address for nonrelocatable load  
JSR FFBA; SETLFS  
LDA \$09 ; number of characters in name  
LDX \$40 ; X and Y contain address of  
LDY \$04 ; filename, in this case at 0440  
JSR FFBD; SETNAM  
LDA \$00 ; 0 for load/1 for verify  
JSR FFD5; LOAD

**Figure 10-6. Machine Language Load (Relocatable)**

LDA \$01 ; logical file number  
LDX \$01 ; device number for tape  
LDY \$00 ; secondary address of zero for relocatable load (tape  
          identifier byte must be 1, identifier of 3 will  
          force nonrelocatable load)  
JSR FFBA ; SETLFS  
LDA \$09 ; number of characters in name  
LDX \$40 ; X and Y contain address of  
LDY \$04 ; filename, in this case at 0440  
JSR FFBD ; SETNAM  
LDA \$00 ; 0 for load/1 for verify  
LDX \$00 ; low byte of address of start of load  
LDY \$60 ; high byte of address of start of load, in this case 6000  
JSR FFD5 ; LOAD

### Figure 10-7. Writing Sequential Files from Machine Language

OPEN for sequential file:

```
LDA $01 ; logical file number
LDX $01 ; device number for tape
LDY $01 ; secondary address of 1 indicates tape write
JSR FFBA ; SETLFS
LDA $09 ; number of characters in name
LDX $40 ; X and Y contain address of
LDY $04 ; filename, in this case at 0440
JSR FFBD ; SETNAM
JSR FFC0 ; OPEN
```

Write to sequential file:

```
LDA $20 ; sample value to write to file
JSR FFD2 ; CHROUT
```

CLOSE the sequential file:

```
LDA $01 ; logical file number
JSR FFC3 ; CLOSE
```

### Figure 10-8. Reading Sequential Files from Machine Language

OPEN for sequential file:

```
LDA $01 ; logical file number
LDX $01 ; device number for tape
LDY $00 ; secondary address of 0 indicates tape read
JSR FFBA ; SETLFS
LDA $09 ; number of characters in name
LDX $40 ; X and Y contain address of
LDY $04 ; filename, in this case at 0440
JSR FFBD ; SETNAM
JSR FFC0 ; OPEN
```

Read 256 (decimal) bytes from sequential file and store at an area starting at 0500:

```
LOOP LDX $00 ; X register not changed by CHRIN
     JSR FFCF ; CHRIN
     STA 0500,X
     INX
     BNE LOOP
```

CLOSE the sequential file:

```
LDA $01 ; logical file number
JSR FFC3 ; CLOSE
```

### Figure 10-9. Saving BASIC Program to Tape

**SAVE** *"filename"*

Optional format:

**SAVE** *"filename", device number, secondary address*

Device number is 1 for tape.

Secondary address is odd for tape identifier of 3 to indicate a nonrelocatable program.

Secondary address is even for tape identifier of 1 to indicate a relocatable program.

Secondary address has bit 1 on (address is 2, 3, 6, 7, etc.) to write an end-of-tape header following the program.

### Figure 10-10. Loading BASIC Program from Tape

**LOAD** *"filename"*

Optional format:

**LOAD** *"filename", device number, secondary address*

Device number is 1 for tape.

Secondary address is zero to allow a program with an identifier byte of 1 to be relocated (normally you won't be able to specify the X and Y registers from BASIC, though, so the secondary address is usually not specified as zero).

Secondary address is nonzero to force a tape to be nonrelocatable no matter what its tape identifier (this nonzero value is recommended if loading machine language tapes for which you don't know the tape identifier).

### Figure 10-11. Verifying BASIC Program from Tape

**VERIFY** *"filename"*

### Figure 10-12. Writing to Sequential Files from BASIC

OPEN the file:

**OPEN** *logical file number, device number, secondary address, "filename"*

Device number is 1 for tape.

Secondary address has bit 0 on (address is 1, 3, 5, etc.) for writing to tape

Secondary address has bits 0 and 1 on (address is 3, 7, 11, etc.) for writing to tape followed by an end-of-tape marker.

Writing a character to the file:

**PRINT#***logical file number, variables*

CLOSE the file:

**CLOSE** *logical file number*

### Figure 10-13. Reading Sequential Files from BASIC

OPEN the file:

**OPEN** *logical file number, device number, secondary address, "filename"*

Device number is 1 for tape.

Secondary address is 0 for reading from tape.

Retrieve a character from the file:

**GET#***logical file number, variable*

Or, to retrieve a variable from the file:

**INPUT#***logical file number, variable*

CLOSE the file:

**CLOSE** *logical file number*

CMD can also be used to write to tape. See Russ Davies' *COMPUTE!'s Mapping the VIC* for a way to use BASIC's SAVE to save areas of memory to tape other than the normal BASIC program.

## Variables Used in Tape Routines

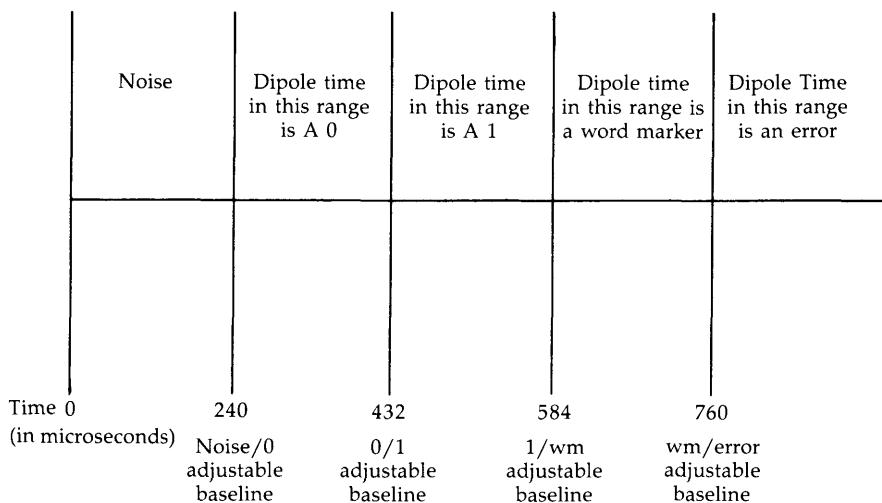
92

0/1 timebase fluctuation (per bit read). The difference between the actual time between FLAG/CA1 interrupts for the first dipole just read (see the description of the routine at F926/F98E for what dipole means) and the adjustable baseline time that

determines whether a dipole is to be considered to have a value of 0 or 1, plus the difference between the actual time for the second dipole just read and the 0/1 adjustable baseline time. 92 is reset to 0 after each bit has been read.

Figure 10-14 demonstrates what is meant by adjustable baseline time and gives the typical equivalent times (for U.S. VIC's with NTSC video) for the various baselines at the start of reading a tape. (The chart is not drawn to scale.)

**Figure 10-14. Adjustable Baseline Concept**



If the resulting value in 92 after reading a bit is greater than or equal to \$80 (128) then for the next bit read, B0 is incremented to add more time to each of the adjustable timebases.

The 0/1 adjustable timebase and the other adjustable timebases are increased if the two dipoles balance to higher value than the 0/1 timebase, while the timebases are decreased if the two dipole balance to a lower value than the 0/1 timebase. (Or the 0/1 timebase line — 76 microseconds if reading leader bits.)

Thus, through comparing the actual time it takes to read a bit to the time the tape routines expect it would take to read this bit, the tape routines are able to make adjustments to their predictions for what values are to be considered noise, 0, 1, or

## Tape I/O Routines

a word marker for the next bit. Through this method, the 64/VIC tape cassette unit can read tapes that were recorded at slightly different speeds than the speed the cassette tape unit is reading. There are limits in tape speed variation beyond which this correction mechanism will not work.

Let's see how this correction might actually work. When reading the all 0 dipoles in the leader (specifically in the leader before the header) the VIC we tested seemed to read a dipole time equivalent to 348 microseconds. Since we're only reading 0 dipoles, we read a dipole of 348 and then another dipole of 348. When computed in F999/F9F3, the resulting value of 92 is \$0C, and thus for the next bit B0 is decremented for less basetime with the 0/1 adjustable timebase equivalent to 428 microseconds. Due to the mathematics involved, location 92 seems to eventually force the adjustable 0/1 timebase value to stabilize to a range of 404–412 microseconds.

However, if the tape is running slow, the adjustable timebases will again stabilize, but this time at higher values.

### 93

During tape read, this flag is read to determine whether a tape load (0) or a verify (1) is being done.

### 96

Indicates during tape load:

If 0, either leader the preceding block has not yet been recognized, or the block has been recognized and now the routine is actually reading data from the block.

If 16–126 (decimal), we have read at least 16 leader bits during read of the tape leader either before the first block (of header or program) or between blocks 1 and 2, and now we're waiting for the word marker at the end of the leader.

### 9B

This is the parity work byte during tape load and save, with bit 0 used to calculate parity. Odd parity is used, where the parity bit is calculated such that the total number of 1's (for all 8 data bits and the parity bit) is an odd number.

### 9C

During tape load:

1 when a byte has been completely received.

0 when waiting for next byte or receiving current byte.



### 9E

Tape save: temporary storage for tape identifier header.

Tape load: Pass 1 error index value; if nonzero then this is two times the number of errors. Indexes into stack where error addresses are stored. 9E limited to maximum value of \$3D, resulting in stack locations 0100–013D possibly used for error storage, a maximum of 31 possible errors.

Tape header load: Index into filename during comparison of filename from tape header to file name specified in LOAD.

Holds the character to be output during CHROUT to tape.

### 9F

Tape load: Pass 2 error correction index, indexes through stack error location addresses. Limited to value no greater than the pass 1 error correction index.

Tape header load: Index into tape buffer during comparison of filename from tape header to file name specified in LOAD.

### A3

Count of bits remaining to be written for a byte (during tape write) or bits remaining to be read for a byte (during tape read). Initialized to 8 before each byte. Decrementing after each bit is written or read.

During tape write, when A3 is decremented to 0, then it's time to set up the parity bit to be written, and when A3 is decremented to -1, it's time to prepare the next byte to be written, so reset A3 to 8.

During tape read, when A3 is decremented to -1, then the parity bit has just been read and it's time to see if the parity bit just read indicated a parity error. After this it's time to read next byte, so reset A3 to 8.

### A4

During tape save: flag to indicate which dipole has been written. After writing a dipole A4 is equal to 1 if we just wrote the first half of the dipole, while it's 0 if we wrote read the second half of the dipole.

During tape load: flag to indicate which dipole has been read. After reading a dipole A4 is equal to 1 if we just read the first half of the dipole, while it's 0 if we just read the second half of the dipole.

### A5

During tape save, counter for block countdown characters that are written to tape before each block's data actually begins. Initialized to 9 for each block, so each block contains 9 block countdown characters. For the first block, the countdown characters have their high bit on, while they don't for the second block. Later, during tape load operations, the block countdown characters can be used to determine whether block 1 or block 2 is being read.

### A6

During CHROUT or CHRIN to tape, A6 is a counter of the number of bytes that have been read from or written to the tape buffer.

### A7

When writing leader dipoles to tape, A7 is used as a counter for an inner loop that must be decremented to 0 each time the loop is performed before the outer loop for writing leader dipoles has its counter decremented. Results in a total of  $A7's\ value \times AB's\ value + 1$  leader dipoles being written. Set to 0 before writing leader for header or program. Set to 80 (decimal) before writing leader between blocks.

During tape load operations, indicates which block is currently being loaded. If A7 contains 2, then the first block is being loaded; if A7 contains 1, second block is being loaded; and if A7 contains 0 then all blocks have been loaded.

### A8

During tape save: switch for word marker write. If A8 contains 0, then write the long time for a word marker dipole. If A8 contains 1 then the long time for a word marker dipole has been written already.

During tape load if nonzero then the byte just read is considered in error. (For example, if a parity error occurs.)

### A9

During tape save: Switch for writing word marker. If A9 contains 0, then write a word marker dipole. If A9 contains 1 then the word marker dipole has been written already.

During tape load: 0/1 balanced counter. Each time a 0 dipole is read, this value is incremented. Each time a 1 dipole is read, this value is decremented.

## Tape I/O Routines

When reading the all 0 leader dipoles and this value reaches 16 (decimal), then 96 is set. The maximum value A9 can be incremented to is 126 (decimal).

When actually reading data bytes, A9 is initialized to 0 before each byte. Since each data bit contains one dipole that is a 0 and the other dipole is a 1, after each bit is read, this counter should be 0 at the end of each bit. If it is not 0, then the byte error flag, B6, is set for this byte.

### **AA**

During tape load this location indicates the action to be taken for the byte just read:

If AA contains 0, then we're waiting for the first block countdown character to arrive. Note: AA is initialized to 0 before read of first header block and before read of first program block.

If AA is from 1-63 (decimal), then block countdown characters are being read.

If AA contains \$40, then valid block countdown characters have arrived, and this byte received is to be treated as a valid data byte.

If AA contains \$80, then the first block has been loaded and we're waiting for the second block.

### **AB**

When writing leader dipoles to tape, AB is used as a counter for an outer loop that must be decremented to -1 before the routine will quit writing leader dipoles. Results in a total of A7 value x AB value + 1 leader dipoles being written. Set to 105 (decimal) before writing leader for header, 20 (decimal) before writing leader for program, and 0 for leader between blocks.

During tape load, once both blocks have been read and the load is considered complete, then compute the parity over all bytes loaded. This parity or checksum should be the same as the checksum that has just been read as the last byte of the tape program or header which was similarly computed over all bytes saved to tape. If not equal, then set a checksum error status. This checksum computation is done both for the header and the program.

The checksum that was recorded during the save at the end of the first block does not appear to be used during the tape load.

### (AC)

During tape save, this pointer (AC) is initialized to the start of the save area, which is (C1), and then after each byte saved it is incremented.

However, each time before a byte is written to tape (right after the word marker long dipole has been written to tape), a check is made to see if AD is greater than or equal to \$80. There are two ways that AD can be greater than or equal to \$80 on the VIC, while just one way on the 64.

One way (both 64 and VIC) is that once the save area and the checksum have been written for a block the high bit is set on for AD through a SEC, ROR AD sequence.

### AD

The other way on the VIC is if you originally specified that you wanted to save from an area greater than or equal to 8000, which would have set (AC) to a value greater than greater than or equal to 8000 and AD greater than or equal to \$80.

When AD is found with its high bit on, then write the interblock leader to tape if the first block was just finished, or turn off the tape motor if the second block was just finished.

Thus, a save from greater than or equal to 8000 on the VIC causes, after a valid header has been written to tape, leader dipoles to be written, specifically it appears the interblock leader would be written, then the tape motor turned off.

Through this dual use of AD on the VIC (both as block end indicator and as the high byte of the save area), it is impossible to save areas on the VIC from greater than or equal to 8000 through the normal tape save routines.

During tape load, this pointer (AC) is initialized to the start of the load area, which is (C1), and then after each byte loaded it is incremented.

### (AE)

Pointer to the end + 1 of the save area. Since when (AC) = (AE) the save is considered complete for this block, and the checksum written, (AE) should be specified as one more than the last byte you want to save. For example, to save the contents of 2000–2500, you should specify a save area of 2000–2501.

Pointer to the end of the load area. LOAD is considered complete for this block when (AC) = (AE).

## Tape I/O Routines

### B0

During tape load, B0 is a factor used in computing what values to set for the adjustable baseline times for the next bit read (see 92 for what adjustable baseline times means).

If  $B0 > 0$ , then more time is added to the adjustable baseline times.

If  $B0 < 0$ , then time is subtracted from the adjustable baseline times.

If  $B0 = 0$ , then no change is made in the adjustable baseline times.

### B1

During tape load: value equivalent to the tape dipole time minus the time between reading timer B/timer 2 and resetting timer B/timer 2. B1 is a one byte field that the two byte timer B/timer 2 value has been compressed into; bits 2–9 of timer B/timer 2 value since timer 2 was last set are stored into bits 0–7 of B1

Temporary variable during timer A/timer 1 set up.

### (B2)

The pointer to the start of the tape buffer with a default of 033C.

### B4

During tape load, is set to nonzero when tape load routines are ready to receive data bytes.

Is reset to zero between blocks.

### B5

During tape load, indicates where tape load routine is currently reading from.

If nonzero, then tape load is before block of data waiting for word marker at the end of the leader bits.

If zero, then actually reading bytes from the block. Set to zero once word marker has been received after 96 has been set.

### B6

During tape load, this flag is set to nonzero if the byte just read was considered in error—either parity error, dipole mismatch, too long dipole, or verify error.

64: During tape save, set to 9 when preparing to write block countdown. During tape load, the end-of-block flag.

## Tape I/O Routines

### **B7**

During tape save: number of characters in filename.

### **B9**

During tape save: if even, then tape identifier recorded is 1 (relocatable program); if odd, then tape identifier recorded is 3 (nonrelocatable program). If bit 1 is on, tape identifier is set to 5 for an end-of-tape header.

During tape load: if reading a tape with tape identifier 3 (nonrelocatable), then secondary address not used as tape is definitely nonrelocatable. If reading a tape with tape identifier 1 (relocatable) then a 0 secondary address allows a relocatable load, a nonzero forces a nonrelocatable load.

### **BD**

During tape save, the byte that is being saved to tape. After one bit is written to tape, the byte is shifted right one bit, and this procedure is repeated until all 8 bits have been written.

During tape load, once the byte has been read it is stored in BD for passing to the byte action routine that handles the byte just read.

### **BE**

During both save and load, the block count.

If BE contains 2, then two blocks remaining to save/load.

If BE contains 1, then one block remaining to save/load.

If BE contains 0, then both blocks have been saved/loaded.

### **BF**

During tape load, the bits read from tape are rotated into BF high to low to build a byte. Once eight bits have been received the byte is considered complete.

### **C0**

Tape motor interlock switch. A nonzero value in C0, which is only possible if some tape buttons are down (as the normal IRQ interrupt handler resets C0 to 0 if no buttons are down), prevents any change of tape motor switch as long as the default IRQ interrupt handler is active.

During tape load or tape save, C0 is set to nonzero once a tape button has been pressed. C0 will be reset to zero once tape load or save are done.

A zero in C0, which is possible with or without buttons down,

## Tape I/O Routines

possibly allows the tape motor to be turned on within the normal IRQ interrupt handler. Possibly allows is used because to turn the tape motor on for the VIC 911C must have bits 2 and 3 set to 1 at entry to the IRQ interrupt handler.

For the VIC, I found that you could turn off the tape motor (outside of the tape routines) by POKE 37148 with any value from 0-7 or 10-11, even if C0 is 0.

For example, try this:

1. Press the Play button on the tape cassette unit. The tape motor is now on.
2. PRINT PEEK(192) displays 0.
3. POKE 37148,0 turns off the tape motor, so C0 (192 decimal) contained a zero, but you can turn the tape motor off and it stays off.

### (C1)

Pointer to the start of the area to be saved or to be loaded.

When loading the header, (C1) is set to point to the start of the tape buffer (B2).

When loading the program, (C1) is set to the same as (C3).

### (C3)

During tape load, contains pointer to the start of the area for a program to be loaded.

If tape identifier of 3 was found in the tape header loaded for this program, then (C3) is taken from the second and third bytes of the tape. (nonrelocatable tape)

If tape identifier of 1 was found in the tape header loaded for this program, then (C3) is taken from the second and third bytes of the tape only if secondary address B9 is nonzero.

If secondary address is zero with tape identifier of 1, then (C3) is set based on the values in the X and Y registers when SAVE was called. (relocatable tape).

### D7

During tape save, the parity over all bytes saved for a block.

During tape load, the value 0 or 1 that the first dipole read represents.

### (029F)

Save area for the IRQ vector during tape load or tape save.

## **Open Logical File for Writing to Tape F3B8/F478-F3D4/F494**

### **Called by:**

BNE at F397/F457 in Determine if Open Is for Read or Write.

When opening a logical file for writing to tape, first prompt for the tape Record and Play buttons to be pressed if no tape buttons were already down.

Prepare a tape identifier of 4, then write to tape a tape buffer with this tape identifier of 4, the starting and ending address of the save area, and the filename.

Then put 2 into the first byte of the tape buffer so that the next buffer written will have the have a tape identifier of 2, indicating a data buffer. Set A6 , count of characters in tape buffer, to 0.

### **Operation:**

1. JSR F838/F8B7 to see if any tape buttons are down. If none are down, then display the PRESS RECORD & PLAY ON TAPE message.
2. If the keyboard stop key is pressed, then exit by BCS to step 6.
3. LDA \$04 and then JSR F76A/F7E7 to write to tape a tape buffer containing the tape identifier of 4 indicating a data file, the starting and ending addresses of the tape buffer, and the filename.
4. With a nonzero (low nybble) secondary address in B9 indicating a write to tape, LDA \$02 and STA in the first byte of the tape buffer. Then set A6 to zero to indicate no characters yet in tape buffer.
5. CLC.
6. RTS.

## **CHROUT to Tape F1DC/F28F-F207/F2B8**

### **Called by:**

Fall through from F1DB/F28E, Determine Output Device; alternate entry at F1DD/F290 by JSR at F2D4/F398 in Close Logical File for Tape.

Character to be written to tape buffer is temporarily stored in 9E.

Increment the count of the number of characters in the



## Tape I/O Routines

tape buffer, A6. Compare to 192 (decimal) to see if the tape buffer is full.

If the tape buffer is full, then set the start and end of the tape buffer from the current pointer to the tape buffer (start) and this value + 192 for the end. Then write the tape buffer to tape. After setting the start and end of the tape buffer put 2 into the first byte of the tape buffer so that the next buffer written will have the tape identifier of 2, identifying the buffer as a data buffer. Set A6 to 1.

After dumping the full buffer to tape, or if the buffer was not full and thus not dumped yet, then store the character in 9E in the next available location in the tape buffer and RTS.

Both the 64 and VIC test to see whether the device is RS-232 (2) or tape (1).

### **Entry conditions:**

A6 contains the number of characters in the tape buffer. (B2) is the pointer to the tape buffer.

### **Exit conditions:**

A6 is incremented if the tape buffer does not yet contain 191 characters. If tape buffer did contain 192 characters, then the buffer dumped to tape and A6 is reset to 1.

The character to be output is placed in next location in the tape buffer.

### **Operation:**

1. Pull the character to be placed in the tape buffer from the stack.
2. F1DD/F290: STA 9E; temporary storage of character to be placed in tape buffer.
3. Save Accumulator, X register, and Y register on stack.
4. JSR F80D/F88A to increment A6, the count of the number of characters in the tape buffer.
5. If there are not yet 192 characters in the tape buffer (including the tape identifier), then branch to step 8.
6. If there are 192 characters in the tape buffer, then JSR F864/F8E3 to write the tape buffer to tape. If the STOP key on the keyboard is pressed, then branch to step 11 with the carry set.
7. Store 2 in the first byte of the tape buffer and reset A6 to 1.
8. LDA 9E, retrieving the character to be placed in the tape buffer.

9. STA in the current location in the tape buffer pointed to by A6.
10. CLC.
11. Restore accumulator, X register, and Y register from the stack.
12. BCC to step 14, thus branching if the STOP key on the keyboard had not been pressed.
13. LDA \$00. If the STOP key is pressed, then exit with accumulator holding 0.
14. RTS.

### **Increment Count of Characters in Tape Buffer F80D/F88A-F816/F893**

#### **Called by:**

JSR at F199/F250 in CHRIN from Tape, JSR at F1E5/F297 in CHROUT to Tape.

Increment the count of the number of characters in the tape buffer, A6. Compare to 192 to see if the tape buffer is full. Return with Z = 0 (BNE condition) if A6 is not equal to 192.

#### **Operation:**

1. JSR F7D0/F84D which loads X and Y registers with address of tape buffer and sees if the tape buffer starts below \$0200. However, it appears this JSR is wasted since nothing is done with the results returned and the subroutine itself does nothing that is used in the CHROUT to tape routines.
2. Increment A6, the count of the number of characters in the tape buffer.
3. Compare A6 to 192. If equal, then Z flag of status register is set to 1 (BEQ condition). If not equal, then Z flag of status register is set to 0 (BNE condition).

### **Set Pointers to Start and End of Buffer and Write Buffer F864/F8E3-F866/F8E5**

#### **Called by:**

JSR at F2D7/E4CF in Close Logical File for Tape, JSR at F1EA/F29C CHROUT to Tape.

Set the start of the tape buffer to be saved to tape (C1) from the current pointer to the tape buffer (B2), and set the

end of the area to be saved (AE) from the start of the tape buffer + 192.

Fall through to F867/F8E6 to write the tape buffer to tape.

### **Entry conditions:**

(B2) is the pointer to tape buffer starting address.

### **Exit conditions:**

(C1) = (B2). (AE) = (B2) + 192.

### **Operation:**

1. JSR F7D7/F854 to set the start and end pointer for the save area from the tape buffer address (B2) for the start and (B2) + 192 for the end.
2. Fall through to F867/F8E6 to write the tape buffer to tape.

## **Close Logical File for Tape F2C8/F38D-F2ED/F3AD**

### **Called by:**

BNE at F2A9/F362 in Determine Device for Close.

Close the logical file for the tape.

If the secondary address is not zero, then we have done a tape write, so do the following:

Store a final byte of 0 in the tape buffer, then write the final tape buffer to tape. Then check to see if the secondary address has bit 1 on, and if so, then write an end-of-tape header with tape identifier of 5.

For either tape read or tape write, then close out the logical file from the file number tables.

### **Operation:**

1. If secondary address is zero, then doing a read, in which case branch to F2F1/F3B1. Otherwise continue with step 2.
2. JSR F7D0/F84D to get the tape buffer address. Again, it appears the results of this subroutine are not used. The 64 also does a SEC which appears unnecessary.
3. LDA \$00, preparing to output a 0 to the tape buffer so that the final byte of a sequential tape file will contain a 0.
4. JSR F1DD/F290 to dump the tape buffer if full and then store 0 as the last byte of the tape buffer.
5. VIC: JMP E4CF.
6. 64: JSR F864 to write this final tape buffer to tape.  
VIC: At E4CF JSR F8E3 to write this final tape buffer to tape.

7. If STOP key pressed then load accumulator with 0.
8. VIC: JMP F39E (step 9) to return control from the patch area back to the main routine.
9. If carry set then stop key was pressed, just RTS.
10. If secondary address has bit 1 on, then LDA \$05 for a tape identifier of 5 and then JSR F76A/F7E7 to write an end-of-tape header.
11. JMP F2F1/F3B1 to close out this logical file in the file tables.

### Control Routine for Tape Save F659/F6F1-F68E/F727

#### Called by:

BCC at F5F8/F690 in Determine Device for SAVE.

This routine controls the tape save operations, calling the routines that write the tape header, the program, and (optionally) an end-of-tape header.

First, check if the device is an RS-232 device, and if it is then jump to the error routine to display illegal device number.

Check if any tape buttons are down (either play, rewind, or fast forward) and if none are down then prompt with message to PRESS RECORD & PLAY.

Display the message SAVING *filename*.

Determine from the secondary address B9 what type of tape is being created, relocatable or nonrelocatable, and set the tape identifier accordingly. An even secondary address in B9 results in a tape identifier of 1 for a relocatable program tape, while an odd secondary address in B9 results in a tape identifier of 3 for a nonrelocatable program.

Jump to subroutine at F76A/F7E7 to fill the tape buffer first with all spaces (\$20), then insert the tape identifier, the starting and ending addresses of the save area, and the filename into the tape buffer. This subroutine at F76A/F7E7 then actually calls the routines that write the tape buffer to tape.

Jump to subroutine at F867/F8E6 to actually write the save area onto tape; first, a short leader (about 3 seconds long) is written onto tape followed by the save area in the format of two identical blocks separated by a short interblock leader.

Finally, if the secondary address has bit 1 on, (e.g., B9 contains \$02 or \$03), then again call the subroutine at

F76A/F7E7 to fill the tape buffer with the tape identifier, starting and ending addresses of the save , and the filename. As before, write this buffer to tape. However, this time the tape identifier used is 5, signifying that this is an end-of-tape header. The tape load routines will not read past an end-of-tape header.

To save a program to tape without having the program preceded by a header, you can JSR to F67C/F715 which will just JSR to F867/F8E6 to write the program to tape, and after the program has been written an end-of-file header can be written if the secondary address in B9 has bit 1 on. If you don't want an end-of-file header or don't want to be concerned about specifying a secondary address, you can just JSR to F867/F8E6.

### Entry conditions:

The accumulator contains the current device number. (C1) contains the starting address of save area. (AE) contains ending address + 1 of save area. B9 contains the secondary address (set by SETLFS). B7 contains the number of characters in the filename and (BB) contains the address of the filename (set by SETNAM).

### Operation:

1. Is current device number (passed in accumulator) = 2?
2. If not, branch to step 4.
3. If yes, then JMP to error routine at F713/F796, which loads accumulator with \$09 (illegal device error) , sets the carry, and RTS.
4. JSR F7D0/F84D to compare the low address of the tape buffer to \$02. (See if tape buffer starting address is below 0200).
5. If low address is less than \$02, then branch to F5F1/F689 which then JMPs to F713/F796 to load accumulator with \$09 (illegal device error), sets the carry, and RTS.
6. If tape buffer low address is greater than or equal to \$02, continue.
7. JSR F838/F8B7 to check tape button status. If neither play, fast forward, or rewind buttons are down on the cassette, then display PRESS RECORD & PLAY message.
8. Branch to step 27 (RTS) if keyboard STOP key was pressed.
9. JSR F68F/F728 to display SAVING and filename.

## Tape I/O Routines

---

10. LDX \$03 to set a possible nonrelocatable tape identifier.
11. LDA with current secondary address in B9.
12. AND \$01. This AND leaves accumulator holding 0 if an even secondary address was specified, while it leaves the accumulator holding 1 if an odd secondary address was specified.
13. BNE to step 15; thus branching if the secondary address was odd, leaving the X register containing \$03, which results in a tape identifier of 3 for a nonrelocatable program tape.
14. LDX \$01. If secondary address was even then X register now contains \$01, which results in a tape identifier of 1 for a relocatable program tape.
15. TXA. The accumulator now contains the tape identifier to put as the first byte of the tape buffer.
16. JSR F76A/F7E7 to fill the tape buffer with the tape identifier, starting and ending addresses of the area to be saved, and the name of the file, then write this tape buffer to tape.
17. If keyboard stop key is down then branch to step 27 (RTS).
18. JSR F867/F836 to write the program to tape.
19. If keyboard stop key is down then branch to step 27 (RTS).
20. LDA with the current secondary address in B9.
21. AND \$02 to determine bit 1 setting.
22. If bit 1 of B9 was 0, then branch to step 25. For example, a secondary address of 1 would cause this path to be taken.
23. If bit 1 of B9 was 1 (for example, secondary address of \$03 or \$02), then LDA \$05 for a tape identifier of 5 indicating this tape header will be an end-of-tape header.
24. JSR F76A/F7E7 to fill the tape buffer with the tape identifier, starting and ending addresses of the area to be saved, and the name of the file, then write this tape buffer to tape.
25. Fall through (using a BIT dummy instruction) to step 27.
26. CLC (also part of BIT dummy instruction)
27. RTS.

### **Load and Check Tape Buffer Address F7D0/F84D-F7D6/F853**

#### **Called by:**

JSR at F2CE/F393 in Close Logical File for Tape, JSR at F3B8/F44B in Open Logical File for Writing to Tape, JSR at F539/F5D1 in Control Routine for Tape LOAD, JSR at F65F/F6F8 in Control Routine for Tape SAVE, JSR at F76C/F7E9 in Prepare Header and Write Buffer to Tape, JSR at F7D7/F854 in Set Start and End of Tape Buffer, JSR at F80D/F88A in Increment Count of Characters in Tape Buffer.

The X register is loaded with the low byte of the address of the tape buffer. The Y register is loaded with the high byte of the address of the tape buffer. Then, to determine whether this address for the tape buffer is valid, the Y register is compared to \$02, which results in the carry clear if the tape buffer high byte address is less than \$02, or the carry set if the tape buffer high byte address is greater than or equal to \$02. The routines that call this routine then can check the carry upon return, and consider it an error if the buffer address is less than 0200.

#### **Entry conditions:**

(B2) contains the address of the tape buffer.

#### **Operation:**

1. LDX with the low address of the tape buffer from B2.
2. LDY with the high address of the tape buffer from B3.
3. Compare this high addresses just loaded into the Y register with \$02; as a result of the compare, the carry is clear if high address less than \$02, while the carry is set if high address is greater than or equal to \$02.

### **Set Start and End of Tape Buffer F7D7/F854-F7E9/F866**

#### **Called by:**

JSR at F7B7/F834 in Prepare Header and Write to Tape, JSR at F847/F8C6 in Read Tape Header into Buffer, JSR at F754/F8E3 in Find Next Tape Header.

JSR to F7D0/F84D to retrieve the tape buffer address in the X and Y registers. Save this address in (C1). Add 192 to this address in (C1) and store the result in (AE) as the end of the tape buffer to be saved or loaded.

## Tape I/O Routines

### Operation:

1. JSR F70D/F84D to retrieve the tape buffer address in the X and Y registers; with X containing the low address and Y the high address.
2. TXA.
3. STA in C1, the low address of the start of the area to be saved or loaded.
4. Add 192 to low address in C1 and store in AE as the low address of the end of the load/save area.
5. TYA.
6. STA in C2, the high address of the start of the area to be saved or loaded.
7. ADC \$00 (so if adding 192 sets the carry above then 1 will be added here).
8. STA in AF, the high address of the end of the load/save area.
9. RTS.

### Prepare Header and Write to Tape F76A/F7E7-F7CF/F84C

#### Called by:

JSR at F2E8/F3A8 in Close Logical File for Tape, JSR at F3BF/F47F in Open Logical File for Writing to Tape, JSRs at F677/F710 and F689/F722 in Control Routine for Tape SAVE.

Fill the tape buffer, pointed to by (B2) with tape identifier, start and end address of area to be saved, and filename.

Temporarily (for this routine only) reset (C1) to the start of the tape buffer and (AE) to the start of the tape buffer + 192.

JSR F86B/F8EA to write a 10 second leader onto the tape, then write the tape buffer onto the tape in the format of two identical blocks separated by a short inter-block leader.

#### Entry Conditions:

The accumulator contains the tape identifier byte. (B2) contains address of tape buffer. (C1) contains the starting address of the save area. (AE) contains the ending address + 1 of the save area.

#### Operation:

1. STA in 9E, temporary storage for the tape identifier.
2. JSR F7D0/F84D to obtain tape buffer address. On return, check to see if the tape buffer starting address is less than



## Tape I/O Routines

0200, and if the buffer is less than 0200 then BCC F7CF/F84C to RTS.

3. Save (C1), the starting address for the save of the program, and (AE), the ending address for the save of the program, on the stack for later restoration. (C1) and (AE) are temporarily reset during this save of the header.
4. Fill all 192 bytes of the tape buffer, which starts at (B2), with spaces (\$20).
5. Retrieve the tape identifier from 9E and put the tape identifier in the first byte of the tape buffer.
6. Put the low address of the start of the save area, C1, in the second byte of the tape buffer.
7. Put the high address of the start of the save area, C2, in the third byte of the tape buffer.
8. Put the low address of the end of the save area, AE, in the fourth byte of the tape buffer.
9. Put the high address of the end of the save area, AF, in the fifth byte of the tape buffer.
10. Set 9F, to be used as index into tape buffer, to 5. Set 9E, to be used as index into filename, to 0.
11. LDY 9E and compare to B7, the number of characters in the filename, and if equal, then branch to step 14. This branch will occur either if the filename contains no characters or when all characters in the filename have been put in the buffer.
12. Get next character of filename and store this character in the next position of the tape buffer.
13. Increment 9E, pointer to filename, and 9F, the pointer to the tape buffer. If 9F has not wrapped around to zero, then branch to step 11. Thus, the limit of the size of a filename is  $256 - 5$ , or 251 characters that could be placed in the tape buffer. A maximum size filename would overlap into the area past the tape buffer.
14. JSR F7D7/F854 to set the start of the save area (C1) to be the same as (B2), the start of the tape buffer. Set the end of the save area  $(AE) = (C1) + 192$ .
15. Set AB, the counter for the outer loop during write of the tape leader, to \$69 (decimal 105).
16. JSR F86B/F8EA to write a 10 second leader to tape followed by the header in two identical blocks (except for the block countdown characters).

17. Restore (C1) and (AE) from the stack to the start of the save area and end of the save area of the program.
18. RTS.

### **Prepare to Write Program to Tape F867/F8E6-F86A/F8E9**

#### **Called by:**

Fall through from F866/F8E5 in Set Pointers to Start and End of Buffer and Write Buffer, JSR at F67C/F715 in Control Routine for Tape SAVE.

Set the outer loop counter for tape leader write, AB, to \$14 (decimal 20). Setting the outer loop counter to this value will cause a total of  $21 \times 256$  cycles of tape leader to be written. Fall through to F86B/F8EA.

#### **Operation:**

1. LDA \$14.
2. STA in AB, the outer loop counter for tape leader write.
3. Fall through to F86B/F8EA.

### **Prepare IRQ Vector and Timer Interrupts for Tape Write F86B/F8EA-F874/F8F3**

#### **Called by:**

Fall through from F86A/F8E9 in Prepare to Write Program to Tape, JSR at F7BE/F83B in Prepare Header and Write to Tape.

Prepare X register for enabling CIA #1 timer B/VIA #2 timer 2 interrupts and accumulator to reset IRQ vector to FC6A/FCA8 for tape write.

#### **Operation:**

1. JSR F838/F8B7 to check tape status line for any tape buttons down. If none are down then display the PRESS RECORD & PLAY message.
2. If keyboard STOP key is down, then BCS to F8DC/F957 to reset 02A0, the saved IRQ vector, to 0, then RTS.
3. Disable IRQ interrupts.
4. LDA \$82/\$A0 to prepare for enabling CIA #1 timer B/VIA #2 timer 2 interrupts in routine at F875/F8F4.
5. LDX \$08 to prepare for indexing into IRQ vector table in routine at F875/F8F4 to reset IRQ vector to FC6A/FCA8.

6. Fall through to F875/F8F4 to set IRQ vector and enable timer B/timer 2 interrupts.

### **Reset IRQ Vector and Set Interrupt Enable Register F875/F8F4-F8CF/F94A**

#### **Called by:**

Fall through from F874/F8F3 in Prepare IRQ Vector and Timer Interrupts for Tape Write, BNE at F862/F8F4 in Load Next Two Blocks to Load Area.

Disable all IRQ interrupts from CIA #1/VIA #2.

If called during save for tape, enable timer B CIA #1/timer 2 VIA #2 interrupts and reset IRQ vector to FC6A/FCA8.

If called during load for tape, enable CIA #1 FLAG/VIA #2 CA1 interrupts and reset IRQ vector to F92C/F98E.

Set BE to 2, the number of blocks to be saved or loaded.

Turn on the tape motor.

CLI so that the next IRQ interrupt will pass control to either FC6A/FCA8 or F92C/F98E.

Loop in the remainder of this routine between IRQ interrupts, testing for the keyboard STOP key and updating the jiffy clock, until the IRQ vector is reset to its value at entry to this routine.

When the next IRQ interrupt occurs after the vector has been reset, then the execution will continue from that new IRQ vector location. Thus, the actual tape save activity starts at FC6A/FCA8, and the actual tape load activity starts at F92C/F98E.

#### **Operation:**

1. Disable all interrupts from CIA#1/VIA #2.
2. Store value of accumulator at entry into DC0D, the CIA #1 interrupt control register/912E, the VIA #2 interrupt enable register. Thus if the accumulator contains \$82/\$A0, timer B/timer 2 interrupts are enabled, while if the accumulator contains \$90/\$82, FLAG/CA1 interrupts are enabled.
3. 64: LDA DC0E, ORA \$19, STA DC0F to force timer B to load its counter from its latched value, set one-shot mode, and start timer B.
4. 64: AND \$91, STA 02A2 to set CIA #2 interrupt log to indicate that both timer A and FLAG interrupts are enabled.

## Tape I/O Routines

It seems that something might be slightly in error here, that this step should have actually appeared after step 2 to correctly reflect which interrupts are enabled for tape I/O.

5. JSR F0A4/F160 to wait for any interrupts from CIA #2/VIA #1 to be serviced and then disabled. Thus, no RS-232 interrupts will interrupt the tape routines.
6. 64: Set bit 4 of D011 to 0, blanking the screen for the tape load or save operation to prevent the VIC-II chip cycle stealing from interfering with the timing of tape I/O.
7. Save the IRQ vector value at entry to this routine (typically EA31/EABF), into (029F), the save area for the IRQ vector during tape processing.
8. JSR FCDB/FCFB to change the IRQ vector based on the X register value. If the X register contains \$08, the new IRQ vector is FC6A/FCA8 for tape leader write; if the X register contains \$0E, the new IRQ vector is F92C/F98E for tape read; if the X register contains \$0A, the new IRQ vector is FB CD/FC0B for tape write; and if the X register contains \$0C, the IRQ vector is reset to its normal value of EA31/EABF.
9. Set BE to 2, indicating the number of blocks to saved or loaded.
10. JSR FB97/FBDB to intialize tape variables: A3 = 8, A4 = 0, A8 = 0, 9B = 0, A9 = 0.
11. Turn the tape motor on.  
64: LDA 01, AND \$1F, STA 01, to set bit 5 to zero, turning the tape motor on.  
VIC: Set bit 1 of 911C, VIA #1 peripheral control register, to 0, and bits 2 and 3 to 1.  
Also, set C0 to this same nonzero value to prevent any change of tape motor setting during default IRQ processing.
12. Execute a delay loop for about .32 seconds to allow the tape motor to gather speed. The loop is as follows (showing the VIC addresses):

Location	Instruction	Cycles
F921	LDX \$FF	2
F923	LDY \$FF	2
F925	DEY	2
F926	BNE F925	3
F928	DEX	2
F929	BNE F923	3

## Tape I/O Routines

For each time the X register is decremented, F925 and F926 are executed 256 times and F923, F928, and F929 are executed once. F921 is only executed once before the entire loop. The loop to F923 is executed 255 times. Total number of cycles is  $255 \times ((256 \times 5) + 7)$ , or a total of 328,185 cycles. The 328,185 cycles divided by 1,022,370 cycles per second on NTSC VIC's gives a total delay of .32 seconds.

13. VIC: Store a value into 9129, the high byte of the VIA #2 timer 2 count, clearing any outstanding timer 2 interrupts.
14. CLI. IRQ interrupts are enabled again now, and the next one will go to FC6A/FCA8 if saving to tape or F92C/F98E if reading from tape. Between execution of the IRQ interrupt handlers for tape I/O steps 15–20 below are executed. These steps continue executing until the IRQ vector has been restored to its value at entry to this routine.
15. LDA 02A0, the low byte of the temporary save area for the IRQ vector during tape save. Now compare this value to the low byte of the currently active IRQ vector in 0315.
16. If 02A0 = 0315, then the tape save or tape load is complete and has reset the default IRQ vector, in which case branch to F8DC/F957 to reset 02A0 to zero and RTS.
17. JSR F8D0/F94B to test for the STOP key and exit if it is detected.
18. VIC: If 02A0 is not equal to 0315, then check VIA #2 interrupt flag register, 912D, to see if any timer 1 interrupts have occurred. If none have occurred, branch to step 15.
19. 64: JSR F6BC to scan the keyboard for the STOP key.  
VIC: If a timer 1 interrupt has occurred, then JSR F734 to increment jiffy clock and test for the keyboard STOP key being down.
20. Branch to step 15.

### Reverse Tape Write Line and Set Timer for Next Interrupt FBA6/FBEA-FBC7/FC05

#### Called by:

JSR at FBF0/FC2E in Write Data Bit to Tape; alternate entry at FBAD/FBF1 by JSR at FBE7/FC25 in Write a Word Marker Bit (this is the entry point for writing a medium dipole); alternate entry at FBAF/FBF3 by JSR at FC6C/FCAA in Write a Leader

## Tape I/O Routines

Bit to Tape; alternate entry at FBB1/FBF5 by JSR at FBD5/FC13 in Write a Word Marker Bit (this is the entry point for writing the word marker dipole).

Flip the tape write line, causing the polarity being written to tape to reverse. Depending on the entry point into the routine set various values for timer B/timer 2 for causing the next timer B/timer 2 IRQ interrupt to occur and flip the tape write line.

### Entry conditions:

If entered at FBA6/FBEA: bit 0 of BD, the byte that is being saved to tape, contains the next bit that is to be saved.

If entered at FBAF/BBF3: The accumulator contains \$78 (120 decimal).

If entered at FBB1/FBF5: The accumulator contains \$10 and the X register contains \$01.

### Exit conditions:

If the tape write line is 1 at exit, set  $Z = 0$  (BNE condition).

If the tape write line is 0 at exit, set  $Z = 1$  (BEQ condition).

### Operation:

1. FBA6/FBEA: LDA BD, the current byte being written to tape. Then LSR to shift bit 0 of this current byte to the carry.
2. LDA \$60, value if carry is clear, if the next bit to be saved is 0.
3. BCC to step 5, if bit 0 of BD was 0.
4. FBAD/BBF1: LDA \$B0. If writing a data bit of 1 or if writing the second dipole of a word marker, then accumulator will be \$B0.
5. FBAF/BBF3: LDX \$00. This entry point from tape leader write has the accumulator set to \$78 before the JSR FBAF/BBF3.
6. FBB1/FBF5: If entry at this point from JSR at FBD5/FC13 when writing the first dipole of the word marker, then accumulator is set to \$10 and X register is set to \$01.

STA in DC06/9128, the low byte for timer B/timer 2.

7. STX in DC07/9129, the high byte of timer B/timer 2. For the VIC, storing this value in 9129, starts timer 2 counting down again. This also resets the timer 2 interrupt flag. For the 64, timer B must be started by storing \$19 in DC0F,

which also forces a load of timer B and sets one-shot running mode. Also on the 64, LDA DC0D to clear CIA #1 interrupt data register.

8. 64: LDA 01, the 6510 I/O port. EOR \$08, STA 01 to flip the value of the tape write.

VIC: LDA 9120, port B I/O register for VIA #2, then EOR \$08 and STA \$9120, in effect flipping the value of the tape write line.

9. AND \$08 so that the status register will have  $Z = 0$  (BNE condition) if the tape write line is 1, or  $Z = 1$  (BEQ condition) if the tape write line is 0.

### **Write Leader Bit to Tape and Reset IRQ Interrupt FC6A/FCA8-FC92/FCCE**

#### **Called by:**

IRQ interrupt (caused by timer B of CIA #1/timer 2 of VIA #2) after the IRQ vector has been reset to FC6A/FCA8.

Write leader cycles to tape. Two cycles (one dipole) are written for each decrement of the inner loop counter A7. Later in this section I discuss how long a cycle lasts. Each time A7 decrements to 0, then decrement the outer loop counter AB. The total number of dipoles written is  $(A7 \times AB) + 1$ . For the leader before a tape header, approximately a 10 second leader is written. For the leader before a program approximately a 3 second leader is written, and between blocks a very short (less than 1 second) leader is written. The leader cycles that are written are very close to the length of time of the values written for 0 data cycles, and two leader cycles are indeed considered a 0 dipole when reading data back from the tape during tape load.

Reset the IRQ vector to FBCD/FC0B and CLI so that the next IRQ interrupt caused by timer B/timer 2 goes to the actual tape data write routine.

However, before this next interrupt occurs, do the following two steps:

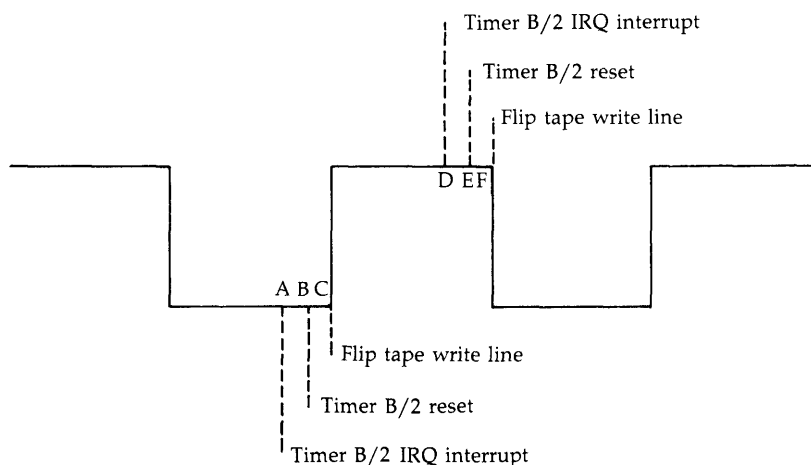
First, test BE to see if both blocks have been saved. If true, then reset the IRQ vector to the default IRQ vector. The way this test to see if both blocks have been saved works is that the tape write routine decrements BE after writing each block, and then resets the IRQ vector to FC6A/FCA8. Thus, at the end of each block, we have a chance to see if we're done with the save.

## Tape I/O Routines

Second, set A5 to indicate we are to write 9 block count-down characters before actually writing the data from the save area onto tape, and then BNE FC16/FC54, jumping right into the middle of the tape write routine to write these block countdown characters.

To determine how many microseconds each leader cycle is written to tape before the polarity reverses (tape write line flipped) for the next cycle, the following calculations were performed. These were based on the VIC but there should be little difference for the 64. There may be some error in these calculations but they should not be too far off. Refer to Figure 10-15 in this discussion.

**Figure 10-15. Leader Cycle**



This figure is not drawn to scale

- A = occurrence of timer B/timer 2 interrupt
- B = reset of timer B/timer 2 to 120
- C = flip of tape write line (timer B/timer 2 now 110)
- D = occurrence of next timer B/2 interrupt
- E = reset of timer B/timer 2 to 120
- F = flip of tape write line (timer B/timer 2 now 110)



## Tape I/O Routines

The time between flips of the tape write line (C-F in Figure 10-15) is equal to the time it takes the current value in timer B/timer 2 (at point C) to countdown to zero (C-D in Figure 10-15), plus the time (D-F in Figure 10-15) between timer B/timer 2 reaching zero and causing an IRQ interrupt and the IRQ interrupt service routine flipping the tape write line. When the tape write line is flipped, timer B/timer 2 has already decremented by 10 for the 10 cycles of instructions between setting the timer and flipping the tape write line.

To summarize the calculations below: a leader cycle takes 174.9 microseconds, and a leader dipole takes 349.8 microseconds.

From the occurrence of the IRQ timer 2 interrupt to the flip of the tape write line the following instructions are executed:

Location	Instruction	Cycles
FF72	PHA	3
FF73	TXA	2
FF74	PHA	3
FF75	TYA	2
FF76	PHA	3
FF77	TSX	2
FF78	LDA \$0104,X	4
FF7B	AND #\$10	2
FF7D	BEQ \$FF82	3
FF82	JMP (\$0314)	<u>5</u>
		29 cycles
FCA8	LDA #\$78	2
FCAA	JSR \$FBF3	6
FBF3	LDX #\$00	2
FBF5	STA \$9128	4
FBF8	STX \$9129	<u>4</u>
		18 cycles
FBFB	LDA \$9120	4
FBFE	EOR #\$08	2
FC00	STA \$9120	<u>4</u>
		10 cycles

Also, there is a fixed 7 cycle delay for IRQ bookkeeping tasks when an IRQ interrupt occurs, plus a variable number of cycles (1-6) for the current instruction to complete (Jim Butterfield, *COMPUTE!* September 1982, p. 156).

## Tape I/O Routines

Thus, the total number of cycles from the occurrence of the IRQ interrupt to the flip of the tape write is  $7 + 29 + 18 + 10 = 64$  cycles (time A-C in Figure 10-15), and  $64 \text{ cycles} \times 1.022370 \text{ microseconds/cycle} = 65.4 \text{ microseconds}$ . To this 65.4 microseconds add the time it takes for timer 2 to count-down to 0 and cause the interrupt. Remember that timer 2's original setting of 120 has already decremented to 110 (time B-C in Figure 10-15) when the tape write line was flipped. Thus,  $(110 + 2)/1,022,370 = 109.5 \text{ microseconds}$  (the time C-D in Figure 10-15). The total time for one cycle of a leader from these calculations should be  $109.5 + 65.4 = 174.9$  microseconds. For a leader dipole (2 cycles) the time should be  $174.9 \times 2 = 349.8 \text{ microseconds}$ .

When I compared this predicted time of a leader dipole of 349.8 microseconds to the actual time I observed upon loading a tape, I found very close agreement as the tape load seemed to be reading leader dipoles of 348 microseconds.

One question that occurs when dealing with the IRQ service routine was the effect of the instruction cycles executed between flipping the tape write line and executing an RTI. Looking at these instructions (from FC03-FCB8, FBDB-FBE9, FC92, and FF56-FF5B on the VIC-20) you can observe that the total number of cycles of the interrupt service routine will not exceed the number of cycles in timer 2, thus allowing the IRQ interrupt service routine to complete and RTI before the next IRQ interrupt caused by timer 2 reaching zero occurs. Thus, nesting of interrupts does not occur.

### **Entry conditions:**

A7 contains 00 before the first block of header or program is written, or 80 (decimal) before the second block is written. AB contains 105 (decimal) before the first block of header is written, 20 (decimal) before the first block of program is written, and 00 before the second block of header or program is written.

### **Operation:**

1. LDA \$78, then JSR FBAF/FBF3 to set timer B/timer 2 to \$0078, and to flip the tape write line. By always setting timer B/timer 2 to the same value when writing the leader, the leader cycles that are written are of the same duration.
2. If the tape write line is 1 then branch to FC54/FC92, which jumps to EEBC/FF56 to restore registers and RTI. Thus, the tape write line is 1 then 0 before each decrement

of the inner loop counter, or one positive and one negative cycle before each decrement.

3. Decrement the inner loop counter A7.
4. If A7 has not reached zero, then branch to FC54/FC92, restoring registers and RTI.
5. JSR FB97/FBDB to initialize tape variables, A3 = 8, A4 = 0, A8 = 0, A9 = 0, 9B = 0.

I didn't see why this JSR was put inside the loop rather than after step 7, although it doesn't do any damage here.

6. Decrement the outer loop counter AB.
7. If AB is greater than or equal to 0 then branch to FC54/FC92 to restore registers and RTI. Thus AB must be -1 to fall through to step 8. For each decrement of AB, A7 is decremented 256 times. The exception to this decrementing of A7 256 times occurs when writing the leader between blocks. After decrementing to -1, AB is incremented, leaving AB = 0 at FC82/FCC0. Thus, between blocks, since AB is not explicitly set to a value as it is before block 1 of the header or program, the first decrement of AB results in it being -1 and thus falling through to step 8. When this between block leader is written, A7 has been set to 80 (decimal), resulting in only 80 dipoles of tape leader values written.

Before the first block of the header, AB contains 105. Thus,  $106 \times 256 = 27,136$  leader dipoles will be written, and  $27,136 \text{ leader dipoles} \times .000349 \text{ seconds/leader dipole} = \text{about a } 9.47 \text{ second leader before the first block of the header.}$

Before the first block of the program is written, AB contains 20. Thus,  $21 \times 256 = 5,376$  leader dipoles will be written, and  $5,376 \text{ leader dipoles} \times .000349 \text{ seconds/leader dipole} = \text{about a } 1.87 \text{ second leader before the first block of the program. This is also the time for the leader before writing the first block of a sequential buffer.}$

8. LDX \$0A to index into IRQ vector table at FD9B/FDF1, then JSR FCBD/FCFB to reset IRQ vector to FBCE/FC0B. However, before the next timer B/timer 2 IRQ interrupt occurs and causes execution to resume at FBCE/FC0B, the following code in steps 9, 10, 11, and 12 is performed.
9. JSR FB8E/FBD2 to reset (AC), the pointer to the current byte to save, to the start of the save area, (C1).

10. Set A5 to 9, indicating that 9 block countdown characters will be written to tape at the start of this block.
11. 64: Set B6 to 9.
12. Branch to FC16/FC54 to start writing these block countdown characters. FC16/FC54 is actually part of the routine that writes the cycles to tape for each dipole.

### **Write Word Marker and Data Bit Cycles to Tape for One Block** **FBCD/FC0B-FC69/FCA7 and** **FBC8/FC06-FBCC/FCDA**

#### **Called by:**

IRQ interrupt (caused by timer B/timer 2 of CIA #1/VIA #2) when IRQ vector has been reset to FBCD/FC0B by the tape leader write routine.

This routine is called with each CIA #1 timer B/VIA #2 timer 2 interrupt to write the cycles onto tape that make up two dipoles for each bit to be saved. The routine also handles writing word markers that precede each data byte written. (During load the word markers are treated as following a byte of data, rather than preceding it.) In describing this routine, I have broken it down into a number of sections with each section treating a separate topic. Keep in mind that all of these sections together make up the IRQ driven tape write routine. Exit this routine by JMP FEBC/FF56 at FC09/FC47 to restore registers and RTI.

The sections consist of:

FBCD/FC0B, write a word marker.

FBF0/FC2E, write a data bit value for 0 or 1.

FBF5/FC33, see which half of the dipole the tape write routine is in.

FBFD/FC3B, prepare for writing the second dipole.

FC0C/FC4A, prepare for next bit to be written from BD.

FC16/FC54, see if need to write a block countdown character.

FC30/FC6E, test if all bytes from save area have been written to tape for this block.

FC3F/FC7D, retrieve next byte from save area and put in BD.

FC4E/FC8C, compute parity bit.

FBC8/FC06, set AD to indicate all of block has been saved.

FC57/FC95, handle end-of-block processing.

To determine how many microseconds each cycle is written to tape before the polarity reverses (tape write line flipped) for the next cycle, the following calculations were performed, using the VIC although the 64 should be similar. (Refer to the Figure 10-15 in these discussions).

The time between flips of the tape write line (C-F in the diagram) is equal to the time it takes the current value in timer B/timer 2 (at point C) to countdown to zero (C-D in the diagram), plus the time (D-F in the diagram) between timer B/timer 2 reaching zero and causing an IRQ interrupt and the IRQ interrupt service routine flipping the tape write line. When the tape write line is flipped, timer B/timer 2 has already decremented by 10 for the 10 cycles of instructions between resetting the timer and flipping the tape write line.

Both the timer B/timer 2 value and the number of instructions that are executed between the occurrence of the timer B/timer 2 interrupt and the flip of the tape write vary when writing data cycles of 0 or 1 or the word marker cycle. Note: I am using the word cycle to mean two different things in this discussion, when talking about 0, 1, or word marker cycles I am referring to the time it takes between reversals of polarity of the tape write line. When giving the number of cycles next to an instruction, I am referring to the number of 6502 clock cycles that instruction takes to execute, or the number of clock cycles that timer B/timer 2 must decrement to reach zero.

To summarize the calculations below (for a U.S. VIC with NTSC video), the 0 cycle takes 172.9 microseconds, the 1 cycle takes 251.2 microseconds, and the word marker cycle takes 329.7 microseconds (for the first one) or 334.9 microseconds (for the second one). A 0 dipole takes 345.8 microseconds, a 1 dipole takes 502.4 microseconds, and a word marker dipole takes 664.6 microseconds.

Notice how close a 0 dipole, 345.8 microseconds, is to a leader dipole, 370.2 microseconds. They are similar for a reason. During tape load only 0 dipoles are read. When 16 of these 0 dipoles have been consecutively read, the 0 dipoles are considered leader dipoles. Otherwise, during the load of a block these dipoles are considered the 0 dipole of a bit of data.

### 0 or 1 Cycles

From the occurrence of the IRQ timer 2 interrupt to the flip of the tape write line the following instructions are executed:

## Tape I/O Routines

Location	Instruction	Cycles
FF72	PHA	3
FF73	TXA	2
FF74	PHA	3
FF75	TYA	2
FF76	PHA	3
FF77	TSX	2
FF78	LDA \$0104,X	4
FF7B	AND #\$10	2
FF7D	BEQ \$FF82	3
FF82	JMP (\$0314)	<u>5</u>
		29 cycles
FC0B	LDA \$A8	3
FC0D	BNE \$FC21	3
FC21	LDA \$A9	3
FC23	BNE \$FC2E	3
FC2E	JSR \$FBEA	<u>6</u>
		18 cycles
FBEA	LDA \$BD	3
FBEC	LSR	2
FBED	LDA #\$60	2 (if A = 0)
FBEF	BCC \$FBF3	3
FBF1	LDA #\$B0	2 (if A = 1)
FBF3	LDX #\$00	2
FBF5	STA \$9128	4
FBF8	STX \$9129	4
FBFB	LDA \$9120	4
FBFE	EOR #\$08	2
FC00	STA \$9120	<u>4</u>
		30 cycles (if A = 0)
		32 cycles (if A = 1)

Also, there is a fixed 7 cycle delay for IRQ bookkeeping tasks when an IRQ interrupt occurs, plus a variable number of cycles (1-6) for the current instruction to complete.

Thus, the total number of cycles from the occurrence of the IRQ interrupt to the flip of the tape write is  $7 + 29 + 18 + 30$  (if 0 cycle) = 84 cycles (time A-C in the diagram). Or if preparing to set timer 2 for a 1 cycle then  $7 + 29 + 18 + 32 = 86$  cycles. However, since a setting of timer 2 for a 0 cycle may follow the second cycle of a 1 dipole, and vice versa, once can't be sure whether the 84 or 86 cycle delay will follow the occurrence of an IRQ interrupt when writing 0 or 1 cycles.

## Tape I/O Routines

Thus, an average of 85 is used.  $85 \text{ cycles} \times 1.022370 \text{ micro-seconds/cycle} = 86.9 \text{ microseconds}$ . To this 86.9 microseconds add the time it takes for timer 2 to countdown to 0 and cause the interrupt. Remember that timer 2's original setting has already decremented by 10 (time B-C in the diagram) when the tape write line was flipped.

For a 0 cycle the timer 2 value set is 96. Thus,  $(86 + 2)/1,022,370 = 86 \text{ microseconds}$  (the time C-D in the diagram). (The + 2 is for the two cycles for reload of the timer once the timer reaches 0.) The total time for one cycle of a 0 from these calculations should be  $86 + 86.9 = 172.9 \text{ microseconds}$ . For a 0 dipole (two cycles) the time should be  $172.9 \times 2 = 345.8 \text{ microseconds}$ .

For a 1 cycle the timer 2 value set is 176. Thus,  $(166 + 2)/1,022,370 = 164.3 \text{ microseconds}$  (the time C-D in the diagram). The total time for one cycle of a 1 from these calculations should be  $164.3 + 86.9 = 251.2 \text{ microseconds}$ . For a 1 dipole (two cycles) the time should be  $251.2 \times 2 = 502.4 \text{ microseconds}$ .

When I compared this predicted time of a 1 dipole of 502.4 microseconds to the actual time I observed upon loading a tape, I found fairly close agreement as the tape load seemed to be reading 1 dipoles of 481 microseconds.

Just as was the case in writing leader dipoles, the interrupt service routine that follows the flip of the tape write line when writing 0, 1, or word marker cycles is not long enough to cause nesting of IRQ timer 2 interrupts.

### Word Marker Cycles

From the occurrence of the IRQ timer 2 interrupt to the flip of the tape write line the following instructions are executed:

Location	Instruction	Cycles
FF72	PHA	3
FF73	TXA	2
FF74	PHA	3
FF75	TYA	2
FF76	PHA	3
FF77	TSX	2
FF78	LDA \$0104,X	4
FF7B	AND #\$10	2
FF7D	BEQ \$FF82	3
FF82	JMP (\$0314)	5
		<hr/> 29 cycles

## Tape I/O Routines

FC0B	LDA \$A8	3
FC0D	BNE \$FC21	3
FC0F	LDA #\$10	2
FC11	LDX #\$01	2
FC13	JSR \$FBF5	<u>6</u>

16 cycles

FBF5	STA \$9128	4
FBF8	STX \$9129	4
FBFB	LDA \$9120	4
FBFE	EOR #\$08	2
FC00	STA \$9120	<u>4</u>

18 cycles

The code above frames the first word marker cycle of timer 2 being set to 272. The first time through the above code, A8 is 0, and the flip of the tape write line frames the previous data cycle, then exits as the tape write line should be 1. The second time through the above code flips the tape write line, framing the cycle time for timer 2 that was set to 272, then finds the tape line is 0, so the program execution continues at FC18 by incrementing A8. Thus, the next time an IRQ timer 2 interrupt occurs (again from a timer 2 setting of 272), then the BNE FC21 is taken, as the code below illustrates.

Location	Instruction	Cycles
FF72	PHA	3
FF73	TXA	2
FF74	PHA	3
FF75	TYA	2
FF76	PHA	3
FF77	TSX	2
FF78	LDA \$0104,X	4
FF7B	AND #\$10	2
FF7D	BEQ \$FF82	3
FF82	JMP (\$0314)	<u>5</u>

29 cycles

FC0B	LDA \$A8	3
FC0D	BNE \$FC21	3
FC21	LDA \$A9	2
FC23	BNE \$FC2E	3
FC25	JSR \$FBF1	<u>6</u>

17 cycles



## Tape I/O Routines

FBF1	LDA #\$B0	2
FBF3	LDX #\$00	2
FBF5	STA \$9128	4
FBF8	STX \$9129	4
FBFB	LDA \$9120	4
FBFE	EOR #\$08	2
FC00	STA \$9120	<u>4</u>
		22 cycles

Also, there is a fixed 7 cycle delay for IRQ bookkeeping tasks when an IRQ interrupt occurs, plus a variable number of cycles (1-6) for the current instruction to complete (Jim Butterfield, *COMPUTE!* September 1982, p. 156).

Thus the total number of cycles from the occurrence of the IRQ interrupt to the flip of the tape write is  $7 + 29 + 16 + 18$  (if first word marker cycle) = 70 cycles (time A-C in the diagram). If second word marker cycle then the number of cycles is  $7 + 29 + 17 + 22 = 75$  cycles. To calculate the length of time required,  $70 \text{ cycles} \times 1.022370 \text{ microseconds/cycle} = 71.5 \text{ microseconds}$ , and  $75 \text{ cycles} \times 1.022370 \text{ microseconds/cycle} = 76.7 \text{ microseconds}$ . To this add the time it takes for timer 2 to countdown to 0 and cause the interrupt. Remember that timer 2's original setting has already decremented by 10 (time B-C in the diagram) when the tape write line was flipped.

For a word marker cycle the timer 2 value set is 272. Thus,  $(262 + 2)/1,022,370 = 258.2 \text{ microseconds}$  (the time C-D in the diagram). The total time for the first word marker cycle should thus be  $71.5 + 258.2 = 329.7 \text{ microseconds}$ . For the second word marker cycle the time should be  $76.7 + 258.2 = 334.9 \text{ microseconds}$ . The total for a word marker dipole should thus be  $329.7 + 334.9 = 664.6$ . This 666.6 microsecond time for a word marker dipole compares to the time I read for a dipole during tape load routines of 676 microseconds. The word marker dipole is made up of two unequal cycles, in contrast to 0 or 1 dipoles each of which contain two equal cycles.

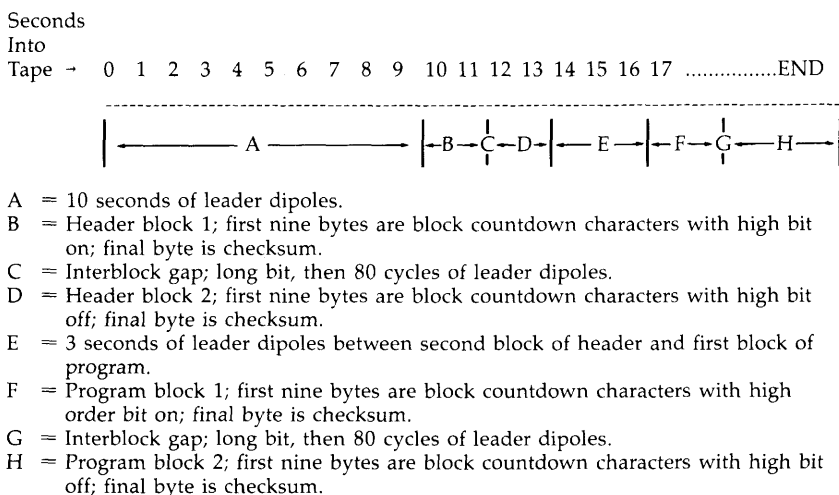
The word marker bit sets timer 2 to 172 (same value as for a data cycle of 1) for writing the two cycles that make up the second dipole of the word marker.

In all of these calculations it should be noted that the clock cycle time of 1.022370 for NTSC (U.S.) VICs was used.

## Tape I/O Routines

Figure 10-16 is a diagram of the typical format of a program saved on tape (viewed from the level of blocks and leaders).

**Figure 10-16. Typical Tape Format**



### Write a Word Marker Bit to Tape FBCD/FC0B-FBEF/FC2D

#### Called by:

IRQ interrupt (caused by timer B/timer 2 of CIA #1/VIA #2) when vector has been reset to FBCD/FC0B by the leader write routine.

Write a word marker bit (which is a word marker dipole, two long cycles, followed by a 1 dipole, two medium cycles).

Then, check B6/AD to see if the high bit is on, indicating block save complete. If so, JMP FC57/FC95 to the end-of-block processing. This test is what prevents saves on the VIC from addresses greater than or equal to 8000, as (AC) uses AD as the high portion of the address to be saved. However, this problem has been fixed on the 64, since a separate byte B6 is used to indicate end-of-block. If this had not been fixed, then large BASIC programs on the 64 that went past 8000 could not have been saved.

## Tape I/O Routines

### Entry conditions:

A8 contains 0 if timer B/timer 2 is to be set to \$0110 for the next tape write cycle, or 1 if timer B/timer 2 has been set to \$0110 twice for two word marker cycles to make up one word marker dipole of a word marker bit. A9 contains 0 if timer B/timer 2 is to be set to \$00B0 for next tape write cycle, or 1 if timer B/timer 2 has been set to \$00B0 twice for two medium (1) cycles to make up the medium dipole of a word marker bit.

### Operation:

Note: In the following instructions I have indicated by a number in parenthesis which time through the routine this instruction will be executed.

1. (4) (3) (2) (1)  
LDA A8. A8 is reset to 0 before the write for each byte.
2. (4) (3) (2) (1)  
BNE to step 12. If timer B/timer 2 has twice been set to 272 then A8 will be 1.
3. (2) (1)  
LDA \$10.
4. (2) (1)  
LDX \$01.
5. (2) (1)  
JSR FBB1/FBF5 to set timer B/timer 2 to \$0110 (272 decimal) and then reverse the polarity on the tape write line.
6. (2) (1)  
Is the tape write line = 1?
7. (1)  
If the tape write line is 1, branch to FC09/FC47 to restore registers and RTI.
8. (2)  
If the tape write line is 0, this routine has just flipped the tape write line at then end of the first word marker cycle.  
INC A8.
9. (2)  
LDA B6/AD. B6/AD has its high bit set to 1 after a block and its checksum have been saved to tape, indicating that this block save is complete.

## Tape I/O Routines

10. BPL to FC09/FC47 to restore registers and RTI, thus branching if high bit is 0.
11. (2)  
    JMP FC57/FC95 to end of block processing. Jump if the high bit was on, if a block save has completed, or (VIC only) if trying to save from greater than or equal to 8000.
12. (3) (4)  
    LDA A9, the flag indicating whether the second half of the word marker dipole has been written yet.
13. (3) (4)  
    BNE FBF0/FC2E. Branch if writing cycle times for actual data bits rather than word marker.
14. (3) (4)  
    JSR FBAD/FBF1. Set timer B/timer 2 to 176, and flip the tape write line.
15. (3) (4)  
    Is the tape write line = 1?
16. (3)  
    If the tape write line is 1, branch to FC09/FC47 to restore registers and RTI, as this routine has just flipped the tape write line at the end of the second word marker cycle.
17. (4)  
    If tape write line is 0, this routine has just flipped the tape write line at then end of the first 1 cycle that makes up the second dipole of the word marker.  
    INC A9.
18. Branch to FC09/FC47 to restore registers and RTI.

### **Write Data Bit to Tape** **FBF0/FC2E-FBF4/FC32**

#### **Called by:**

Fall through from FBEF/FC2D in Write Word Marker to Tape, BNE at FBE5/FC23 in Write a Word Marker to Tape.

Call FBA6/FBEA to set timer B/timer 2 to 176 if writing a 1 or 96 if writing a 0, then flip the tape write line, with the tape polarity cycle time for this flip of the tape write line determined by the previous setting of timer B/timer 2. Exit from the tape write routines through an RTI if the tape write

line is on, which it is the first time through. Thus, two tape polarity cycles of the same time duration are written for each dipole, although the second cycle is actually framed by a flip of tape write line for the next timer B/timer 2 interrupt.

### **Entry conditions:**

A8 contains 1 if timer B/timer 2 has been set to \$0110 twice for two word marker cycles to make up one word marker dipole of a word marker bit.

A9 contains 1 if timer B/timer 2 has been set to \$00B0 twice for two medium (1) cycles to make up the medium dipole of a word marker bit.

Bit 0 of BD contains the next bit to be written to tape. When writing the second dipole for this bit, the value in bit 0 has been reversed, as each bit is represented by dipoles of 0-1 or 1-0.

### **Operation:**

1. JSR FBA6/FBEA. FBA6/FBEA shifts bit 0 of BD into the carry. If the carry is clear, then a 0 cycle is being written for the next tape polarity cycle, in which case set timer B/timer 2 to 96. If the carry is set, then a 1 cycle is being written for the next tape polarity cycle, consequently, set timer B/timer 2 to 176.

Then reverse the polarity being written to tape by Exclusive OR of the tape write line, bit 3 of 01/9120.

2. On return from the JSR, if the tape write line is 1, then branch to FC09/FC47 to restore registers and RTI. If the tape write line is 0, then fall through to FBF5/FC33.

## **Determine Which Part of Dipole Tape Write Routine Is Executing**

**FBF5/FC33-FBFC/FC3A**

### **Called by:**

Falls through from FBF4/FC32 in Write Data Bit to Tape.

See which half of the dipole the tape write routine is executing. If A4 contains 0 at entry then we have just finished writing the first dipole, in which case reset A4 to 1, then fall through to FBFD/FC3B to prepare for second half of dipole. If A4 contains 1 at entry, then we have just finished writing the

second dipole (completed a bit), in which case reset A4 to 0, the branch to FC0C/FC4A to prepare for writing the next bit.

### **Operation:**

1. LDA A4, then EOR \$01 and store the result back in A4.  
Thus, if A4 was 0 at entry to step 1, A4 contains 1 at exit from step 1, indicating the first dipole has been written. If A4 was 1 at entry to step 1, A4 contains 0 at exit from step 1 as the second dipole has been written (bit completed). A4 is initialized to zero before writing each byte.
2. BEQ FC0C/FC4A. If A4 contains 0 at exit from step 1, then tape write routines are finished writing the tape polarity cycles for this bit; it's time to move on to the next bit. This test of A4 makes certain that each bit has both a tape polarity on, tape polarity off sequence.
3. If A4 is not zero at exit from step 1, then fall through to FBFD/FC3B to prepare for writing the second dipole for this bit.

### **Prepare to Write Second Dipole for this Bit FBFD/FC3B-FC0B/FC49**

#### **Called by:**

Falls through from FBFC/FC3A in Determine Which Part of Dipole Tape Write Routine Is Executing.

Prepare for writing the second dipole for this bit. Exclusive OR (flip) the value of the bit just written (bit 0 of BD) to force the second dipole to write a tape polarity that is the reverse of the first dipole polarity. For example, if the first dipole wrote cycles for values of 1-1, then for the second dipole the tape routines write cycles of 0-0. Conversely, if the first dipole contained cycles of 0-0, then the second dipole contains cycles of 1-1. Thus, a bit value of 1 is represented by four cycles of 1-1-0-0, a 0 is represented by four cycles of 0-0-1-1, and a leader by 0-0-0-0. Bits are written to tape with four cycles of alternating tape polarity, while bits are read from tape as two dipoles.

#### **Operation:**

1. LDA BD, then EOR \$01 and store the accumulator back into BD, thus flipping the value that will be written for the second dipole.
2. AND \$01 so only bit 0 is significant.

3. EOR 9B and STA 9B. The parity work byte, 9B is set to zero before writing each byte. Steps 2 and 3 of this routine adjust the current parity work byte by an exclusive or of the current state of the parity byte 9B with the reverse of the current bit being written. Odd parity (total number of bits including parity bit that have value of 1 is odd) is used in both writing and reading tape for the 64 and VIC.
4. JMP FEBC/FF56 to restore registers and RTI.

### **Prepare to Write Next Bit and Decrement Bit Counter FC0C/FC4A-FC15/FC53**

#### **Called by:**

BEQ at FBFB/FC39 in Determine Which Part of Dipole Tape Write Routine Is Executing.

Shift BD right one bit, moving the next bit to be sent into bit position 0.

Decrement and test A3. If ready to do parity bit then execute FC4E/FC8C. If done with all 8 data bits and the parity bit, then prepare for the next byte to send by falling through to FC16/FC54 to reset variables and counters. If A3 is greater than 0 after being decremented, then more bits remain to be written from the current byte, in which case just restore registers and RTI.

#### **Operation:**

1. LSR BD, shift the next bit to be written in bit position 0.
2. Decrement A3.
3. If A3 is equal to 0 then all 8 data bits have been written to tape for this byte; BEQ FC4E/FC8C to calculate the parity bit to be written.
4. If A3 is greater than 0 then more bits from this byte remain to be written to tape. Just branch to FC09/FC47 to restore registers and RTI.
5. If A3 is less than 0 then all 8 data bits and the parity bit have been written to tape for this byte, so fall through to FC16/FC54 to prepare for the next byte to be written.

## Prepare Counters for Next Byte and Test if Writing Block Countdown Characters FC16/FC54-FC2F/FC6D

### Called by:

Fall through from FC15/FC53 in Prepare to Write Next Bit and Decrement Bit Counter, BNE at FC91/FCCD in Write Leader Bit to Tape and Reset IRQ Interrupt.

Reset the counters and variables for the next byte to be saved to tape.

Test A5 to see if any more block countdown characters are to be written for this block. If so, then write the current block countdown character to tape with its high bit turned on if this is the first block, or turned off if this is the second block.

### Entry conditions:

A5 contains 0 when all block countdown characters have been written for this block, or 9-1 when writing block countdown characters for this block.

### Operation:

1. JSR FB97/FBDB to reset counters and variables used when writing bytes to tape: A3 = 8, A4 = 0, A8 = 0, A9 = 0, 9B = 0.
2. CLI. During this IRQ interrupt service routine if another IRQ interrupt occurs past FC19/FC57, then that interrupt will be serviced and the current one being executed will be nested. Nesting of interrupts should not occur as the FBCD/FC0B tape write interrupt service routine takes less cycles to execute than the number of cycles for which timer B/timer 2 has to count down to zero and cause an interrupt.
3. LDA A5, the counter for block countdown characters.
4. If A5 = 0, branch to FC30/FC6E, as all block countdown characters have already been written for this block.
5. If A5 is not zero, first reset D7 to 0 for computing the checksum for this block. Then determine which block is being written. If block one, then the accumulator (which contains value of A5) is ORA \$80 to turn on the high bit. If block 2 is being written, the high bit remains zero. Later, during tape load, the high bit is checked to determine which block is being loaded.
6. Finally, STA BD to prepare for writing this block countdown character to tape, then restore registers and RTI.



## Check for End of Tape Save FC30/FC6E-FC3E/FC7C

### Called by:

BEQ at FC1C/FC5A in Prepare Counters for Next Byte and Test if Writing Block Countdown Characters.

See if all bytes from save area have been written to tape. If not, then branch to FC3F/FC7D.

If the last character from the save area was just written to tape, then now (AC) = (AE). Store the checksum D7 into BD as the last byte to be written to tape for this block.

If all bytes from the save area and the checksum have been written to tape, then branch to FBC8/FC06 to handle end of block processing.

### Entry conditions:

(AC) < (AE) if more bytes remain to be written to tape from the save area.

(AC) = (AE) if all byte have been written from the save area, but the checksum has not yet been written.

(AC) > (AE) if all bytes from the save area and the checksum have been written to tape.

### Exit conditions:

If (AC) = (AE) at entry, AD incremented and D7 stored in BD.

### Operation:

1. JSR FCD1/FD11 to subtract (AE) from (AC).
2. If carry is clear (borrow generated during subtraction), then (AE) > (AC) and more bytes remain to be saved. Branch to FC3F/FC7D.
3. If carry is set (no borrow generated during subtraction) on return, then also check to see if (AC) = (AE). If not equal, then the checksum has been written to tape and AD incremented; branch to FBC8/FC06 to do end of block processing.

- The first time that the carry is set on return from JSR FCD1/FD11, these two pointers (AC) and (AE) should be equal. If the pointers are equal, then fall through to step 4.
4. All of the bytes from the save area have been written to tape. Now it's time to prepare the checksum to be written, and to set AD so that at the end of writing the checksum step 3 will branch to the end of block processing.

Increment AD, the low address of the save area, so that the next time the JSR FCD1/FD11 is executed in step 1 the carry status will return with the carry set, and will also return with  $Z = 0$  (the BNE condition).

Transfer D7 to BD to prepare for having the checksum be the last byte written for this block.

5. Branch to FC09/FC47 to restore registers and RTI.

### **Move Next Byte from Save Area and Increment Pointer**

**FC3F/FC7D-FC4D/FC8B**

#### **Called by:**

BCC at FC33/FC71 in Check for End of Tape Save.

Load the next byte from the save area and store this byte into BD, the next byte to be written to tape.

Exclusive or this byte with the value of the checksum D7 and store the result back into D7. Thus parity over all bytes saved for a block is calculated.

Increment the pointer to the save area, (AC).

#### **Operation:**

1. Load next byte from save area pointed to by (AC) into accumulator, then store the accumulator into BD, the next byte to be written to tape.
2. Exclusive OR the accumulator with D7, storing this value back into D7 to update the parity over all bytes saved.
3. JSR FCDB/FD1B to increment the pointer to the save area (AC).
4. Branch to FC09/FC47 to restore registers and RTI.

### **Prepare Parity Bit for this Byte**

**FC4E/FC8C-FC56/FC94**

#### **Called by:**

BEQ at FC12/FC50 in Prepare to Write Next Bit and Decrement Bit Counter.

Compute the parity bit for the byte just written and store in bit 0 of BD to allow the parity bit for the this byte to be written to tape.

Let's work through an example.

If a byte to be written was \$25, then the binary value is 00100101. However, the parity work byte is computed from

## Tape I/O Routines

the flipped value of the bit for the second dipole. Thus the actual values in the accumulator that will be Exclusive ORed with 1 (EOR \$01) will be: 11011010.

9B is initialized to zero before each byte. Thus, for a data byte of \$25, the parity bit is calculated as follows:

Accumulator:	1 1 0 1 1 0 1 0
9B:	<u>0 1 0 0 1 0 0 1</u>
Accumulator EOR 9B	1 0 0 1 0 0 1 1

Thus, the final result in 9B after eight data bits is 1.

To compute the parity bit:

LDA 9B	1
EOR \$01	<u>1</u>
Accumulator	0

Finally, STA BD, the parity bit for this byte.

Thus for this byte of \$25, a total of three ones (including the parity bit) are stored on tape. Thus the parity bit calculation makes certain that an odd number of ones will be written for a byte, including the parity bit. On loading a tape, if the tape read routines find an even number of ones, then a parity error has occurred.

### Entry conditions:

9B, the parity work byte, contains the value of the parity bit computed for the 8 data bits.

### Operation:

1. LDA 9B, the parity bit calculation for the eight data bits.
2. EOR \$01 to insure that odd parity is being used.
3. STA BD. Store the parity bit into bit position 0, to be the final bit to be written to tape for this byte.
4. FC54/FC92: JMP FEBC/FF56 to restore registers and RTI.

### Indicate Block Save Complete

**FBC8/FC06-FBCC/FC0A**

### Called by:

BNE at FC35/FC73 in Check for End of Save.

Set B6/AD to have its high bit on to indicate the end of block has been reached.

**Operation:**

1. SEC.
2. ROR B6/AD. This ROR rotates the carry into the high bit, and since the carry was just set, a 1 will be rotated into this high bit.
3. BMI FC09/FC47 to restore registers and RTI.

**Handle End-of-Block Processing and Reset IRQ Vector FC57/FC95–FC69/FCA7****Called by:**

JMP at FBE0/FC1E in Write Word Marker Bit to Tape.

When a block has been saved (or if trying to save from greater than or equal to 8000 on the VIC) then this routine is executed.

If the second block was just saved, then turn off the tape motor. If the first block was just saved, write a word marker dipole and then 80 leader cycles.

Reset the IRQ vector to FC6A/FCA8, thus passing control back to FC6A/FCA8 at the end of saving each block.

If you do try to save from greater than or equal to 8000 on the VIC, then it will appear externally from the screen messages and the tape movement that something is being saved to tape. All that is being saved is the header, which correctly points to the save area greater than or equal to 8000, but then this is followed by leader cycles.

**Entry conditions:**

BE contains 2 if the first block of header or program was just saved, or 1 if the second block of header or program was just saved.

**Exit conditions:**

BE contains 1 if the first block of header or program was just saved, or 0 if the second block of header or program was just saved. (0314), the IRQ interrupt vector, is reset to FC6A/FCA8. The tape motor is turned off if the second block of header or program was just saved.

**Operation:**

1. Decrement BE.
2. If BE is nonzero after the decrement then the first block of the header or the program was just saved; branch to step 4.

3. If BE is zero after the decrement then the second block of the header or the program was just saved; JSR FCCA/FD08 to turn off the tape motor. With the tape motor off the leader cycles written in step 4 can't be written after the second block.
4. Set A7 to 80, for 80 cycles of leader dipoles between blocks.
5. Disable IRQ interrupts to prevent an interrupt from occurring while the IRQ vector is being reset.
6. JSR FCBD/FCFB to reset the IRQ interrupt to FC6A/FCA8. Thus, at the end of each block, control returns to FC6A/FCA8.
7. Branch to FC54/FC92 to restore registers and RTI.

### **Check for Tape Button Down**

**F82E/F8AB-F837/F8B6**

#### **Called by:**

JSRs at F817/F894 and F824/F8A1 in Display PRESS PLAY ON TAPE, JSR at F838/F8B7 in Display PRESS RECORD & PLAY ON TAPE; alternate entry at F836/F8B5 by BEQ/JSR at F81A/F897 in Display PRESS PLAY ON TAPE, BEQ/JSR at F83B/F8BA in Display PRESS PLAY & RECORD ON TAPE.

Check bit 4 of 01/bit 6 of 911F, the tape switch line, to see if any tape buttons (Fast Forward, Play, or Rewind) are down.

Return with Z = 0 (BNE) if no tape buttons are down, or with Z = 1 (BEQ) if a tape button is down.

If you press Play rather than Play and Record, the tape save routines won't be able to tell that Record is not down and will act as if everything is okay. However, since 10 seconds of leader are written to tape before the header is saved, the tape save works as long as you press Record in time to allow enough zero dipoles to be written to tape to allow the tape load to function correctly. That is, if you specify a tape save and accidentally just press Play rather than Play and Record, go ahead and press Record rather than starting the save all over. Just press Record in enough time to write at least 16 leader dipoles (that's within about 9.5 seconds, or 16 leader dipoles of 349 microseconds/dipole).

#### **Operation:**

1. LDA \$10/\$40 to prepare to test bit 4 of 01/bit 6 of 911F.
2. BIT 01/911F to test the tape switch line.

3. If tape switch line was 1 (no tape buttons down) then branch to step 5, with  $Z = 0$  (BNE condition).
4. If bit 6 was 0 (tape buttons down), then BIT 01/911F to again set  $Z = 1$  (BEQ condition). I do not see why the BIT 01/911F instruction is repeated, since the carry is cleared in step 5 in either case and the Z bit of the status register will be set the same for this second BIT. Only if the tape button is released in the very short time between steps 2 and 4 will the second BIT return a different value than the first BIT.
5. F836/F8B5: CLC.
6. RTS.

### **Display PRESS PLAY ON TAPE F817/F894-F82D/F8AA**

#### **Called by:**

JSR at F399/F459 in Open Logical File for Reading from Tape, JSR at F541/F5D9 in Control Routine for Tape LOAD, JSR at F84A/F8C9 in Load Next Two Blocks; alternate entry at F81E/F89B by BNE at F83F/F8BE in Display PRESS RECORD & PLAY ON TAPE.

See if any tape buttons are down. If none are down then display either PRESS PLAY ON TAPE if entered at F817/F894, or PRESS PLAY & RECORD ON TAPE if entered at F81E/F89B.

If any tape buttons are down, then branch to just CLC and RTS.

If no buttons are down, check if the keyboard STOP key is pressed, and if it is, then RTS.

Again check if any tape buttons are down. If none are down, then branch to the previous step to check the keyboard STOP key. If any tape buttons are down, then display OK and exit.

#### **Operation:**

1. JSR F82E/F8AB to test the tape switch line to determine if any tape buttons are down. If Play, Fast Forward, or Rewind are down, then BEQ to F836/F8B5 to CLC and RTS.
2. If no tape buttons are down, then LDY \$1B to index the message PRESS PLAY ON TAPE.
3. F81E/F89B: JSR F12F/F1E6 to display the message indexed by the Y register.

4. JSR F8D0/F94B to see if the keyboard STOP key is down. If so, RTS from this routine.
5. JSR F82E/F8AB to test the tape switch line to determine if any tape buttons are down. If none are down, then branch to step 4.
6. If any tape buttons are down, then LDY \$6A and JSR F12F/F1E6 to display message OK.

### **Display PRESS RECORD & PLAY ON TAPE F838/F8B7-F840/F8BF**

#### **Called by:**

JSR at F3B8/F478 in Open Logical File For Writing To Tape, JSR at F664/F6FD in Control Routine for Tape Save, JSR at F86B/F8EA in Prepare IRQ Vector and Timer Interrupts for Tape Write.

JSR to F82E/F8AB to see if any tape buttons are down. If any are down, then CLC and RTS.

If no tape buttons are down, then display PRESS RECORD & PLAY ON TAPE, wait for a tape button to be pressed, and display OK.

#### **Operation:**

1. JSR F82E/F8AB to see if any tape buttons are down.
2. If any tape buttons are down, then BEQ to F836/F8B5 to CLC and RTS.
3. If no tape buttons are down, then LDY \$2E to index the message PRESS RECORD & PLAY ON TAPE.
4. Branch to F81E/F89B to display the message, test for tape buttons down or keyboard STOP key pressed, and display OK if tape button pressed.

### **Check Keyboard STOP Key During Tape I/O F8D0/F94B-F8E1/F95C**

#### **Called by:**

JSR at F821/F89E in Display PRESS PLAY ON TAPE, JSR at F8C7/F938 in Reset IRQ Vector and Set Interrupt Enable Register.

See if the keyboard STOP key is pressed. If it is not down, then just RTS.

If the STOP key is down, then JSR FC93/FCCF to restore certain CIA #1/VIA #2 registers and reset the IRQ vector at

(0314) to the value that was saved in (029F). Then store zero in 02A0.

The carry is set on exit from routine if the STOP key is down.

If the STOP key is down, then rather than doing a normal RTS, the current return address is pulled from the stack. Thus, when the RTS for this routine occurs, control returns not to the routine that called F8D0/F94B, but to the routine that called the routine that called F8D0/F94B. Thus, control does not return to the instruction following the JSR F8D0/F94B, but to the instruction following the previous JSR.

### **Operation:**

1. JSR FFE1, the Kernal STOP vector, to test the STOP key, which JMPs (0328) to the default test STOP key routine at F6ED/F770. If the STOP key is pressed, then the Z = 1 (BEQ condition) on return.
2. CLC in case the STOP key is not pressed.
3. BNE to step 8 to RTS if the STOP key is not pressed.
4. If the STOP key is pressed, then JSR FC93/FCCF to restore CIA #1/VIA #2 registers and reset IRQ vector.
5. SEC to indicate the stop key is pressed; carry flag can be tested on return to see if STOP key pressed.
6. Pull the next two bytes off the stack, thus removing the current return address and forcing the previous two bytes to be used as the return address for the following RTS.
7. Reset 02A0, high byte of IRQ vector temporary storage, to zero.
8. RTS.

### **Reset Pointer to Start of Load/Save Area FB8E/FBD2-FB96/FBDA**

#### **Called by:**

JSR at F617/F6AF Save to Serial Device, JSR at FAB1/FAFE in Check for Last Block Countdown Character, JSR at FB6B/FBAF in Tape Load Completed, JSR at FC88/FCC6 in Write Leader Bit to Tape and Reset IRQ Interrupt.

Reset the pointer to the current byte for the load or save area (AC) from the value in (C1), the pointer to the start of the save area or the start of the load area.

#### **Operation:**

1. Store value from (C1) into (AC), then RTS.



### **Reset Counters and Variables for Tape I/O FB97/FBDB-FBA5/FBE9**

#### **Called by:**

JSR at F8A8/F912 in Reset IRQ Vector and Set Interrupt Enable Register, JSR at FA60/FAAD in Determine Action to Take for this Byte, JSR at FC16/FC54 in Prepare Counters for Next Byte and Test if Writing Block Countdown Characters, JSR at FC75/FCB3 in Write Leader Bit to Tape and Reset IRQ Interrupt.

Reset counters and variables used in writing or reading a byte during tape I/O.

#### **Operation:**

1. Set A3 to 8.
2. Set A4, A8, 9B, and A9 to 0.

### **Reset CIA/VIA Registers and Restore IRQ Vector FC93/FCCF-FCB7/FC5F**

#### **Called by:**

JSR at F8D6/F951 in Check Keyboard STOP Key During Tape I/O, JSR at FB68/FBAC in Tape Load Completed, JSR at FCB8/FCF6 in Set IRQ Vector.

Now that tape I/O operations are complete, turn off the tape motor. Disable IRQ interrupts. Reset CIA #1 interrupt control register/VIA #2 interrupt enable register to disable all CIA #1/VIA #2 interrupts. For the VIC, reset keyboard scan for column 3 and set the VIA #2 auxiliary control register to default of \$40. Enable CIA #1 timer A/VIA #2 timer 1 interrupts and reset CIA #1 timer A value/VIA #2 timer 1 value. If 02A0, high byte of IRQ vector temporary storage, is nonzero then reset (0314) from (029F). On the 64, make the screen visible again.

#### **Operation:**

1. Save status register on stack.
2. Disable IRQ interrupts.
3. 64: Store \$10 in D011, VIC-II chip control register, to turn off the blank screen bit, making the screen visible again.
4. JSR FCCA/FD08 to turn off tape motor.
5. 64: Store \$7F in DC0D to disable all CIA #1 interrupts.  
VIC: Set VIA #2 interrupt enable register to disable all VIA #2 interrupts.

6. VIC: Reset 9120 Port B I/O Register to scan for column 3.
7. VIC: Set 912B VIA #2 Auxiliary Control Register to \$40: output to PB7 disabled, free running mode for timer 1, one-shot interval mode for timer 2, shift register disabled, no latching for I/O Ports A and B.
8. 64: JSR FDDD to enable timer A interrupts for CIA #1, re-set the timer A value, and start the timer.  
VIC: JSR FE39 to enable timer 1 interrupts for VIA #2 and to reset timer 1 value.
9. If 02A0 is nonzero, indicating that (0314) has not yet been reset, then reset (0314), the IRQ vector, from the value in (029F).
10. Pull status register from stack and RTS.

### Set IRQ Vector

**FCB8/FCF6-FCC9/FD07**

#### Called by:

BEQ at FC86/FCC4 in Write Leader Bit to Tape and Reset IRQ Interrupt; alternate entry at FCBD/FCFB by JSR at F8A1/F90B in Reset IRQ Vector and Set Interrupt Enable Register, JSR at FC65/FCA3 in Handle End-of-Block Processing and Reset IRQ Vector, JSR at FC7E/FCBC in Write Leader Bit to Tape and Reset IRQ Interrupt.

If normal entry point, then just JSR FC93/FCCF to restore CIA #1 registers/VIA #2 registers and set (0314) from value in (029F). On the 64 also make the screen visible again. Then exit by branch to FC54/FC92 to restore registers and RTI.

If entry at FCBD/FCFB, then set the IRQ vector (0314) from a table at FD9B/FDF1, with the table entry indexed off of the base of FD93/FDE9, FD94/FDEA using the X register as an index.

#### Exit conditions:

- (0314) = F92C/F98E if X register = \$0E.
- (0314) = FC6A/FCA8 if X register = \$08.
- (0314) = FB CD/FC0B if X register = \$0A.

#### Operation:

1. JSR FC93/FCCF to restore CIA #1/VIA #2 registers and set IRQ vector (0314) from (029F) if it hasn't already been reset.
2. Since FC93/FCCF routine does not affect the status register,

the BEQ that was true when step 1 was called is still true, so BEQ FC54/FC92 to restore registers and RTI.

3. FCBD/FCFB: Use FD93/FDE9 as base and X register as index to retrieve the low address of the new IRQ vector and store this in 0314.
4. Use FD94/FDEA as base and X register as index to retrieve the high address of the new IRQ vector and store this in 0315.
5. RTS.

### **Turn Off Tape Motor** **FCCA/FD08-FCD0/FD10**

#### **Called by:**

JSR at FC5B/FC99 in Handle End-of-Block Processing and Reset IRQ Vector, JSR at FC9D/FCD1 in Reset CIA/VIA Registers and Restore IRQ Vector.

For the 64, set bit 5 of 01, 6510 I/O port, to 1 to turn off tape motor. For the VIC, set bits 3-1 of 911C, VIA #1 peripheral handshaking control register, to 1, turning the tape motor off by holding CA2 high.

#### **Operation (64):**

1. LDA 01, 6510 I/O port.
2. ORA \$20.
3. STA 01.

#### **Operation (VIC):**

1. LDA 911C, VIA #1 peripheral handshaking control register.
2. ORA \$0E.
3. STA \$911C.

### **Compare Pointer to Current Byte with Pointer for End of Load/Save** **FCD1/FD11-FCDA/FD1A**

#### **Called by:**

JSR at F624/F6BC in Save to Serial Device, JSR at FACE/FB1B Check for End of Tape Load, JSR at FB78/FBBF Tape Load Completed, JSR at FC30/FC6E Check for End of Tape Save.

Subtract (AE), the pointer to the end of the load/save area from (AC), the pointer to the current byte being loaded/saved. If a borrow is required because (AE) is greater than (AC) then the carry will be clear at exit. If a borrow is not

required because (AE) is less than or equal to (AC) then the carry will be set at exit. The carry status can then be checked by the calling routine to determine if the load/save is complete.

### **Entry conditions:**

(AE) is the pointer to the end+1 of the load/save area. (AC) is the pointer to the current byte being loaded/saved.

### **Exit conditions:**

Carry clear if (AE) is greater than (AC) (save/load not yet done).

Carry set if (AE) is less than or equal to (AC) (save/load complete).

### **Operation:**

1. SEC.
2. LDA AC, the low byte of the pointer to the current byte being saved/loaded.
3. SBC AE, the low byte of the pointer to the end of the save/load area.
4. LDA AD, the high byte of the pointer to the current byte being saved/loaded.
5. SBC AF, the high byte of the pointer to the end of the save/load area.
6. RTS.

## **Increment Pointer to Current Byte FCDB/FD1B-FCE1/FD21**

### **Called by:**

JSR at F63A/F6D2 in Save to Serial Device, JSR at FB43/FB90 in Load Byte and Increment Pointer to Load Area, JSR at FB78/FBBC in Tape Load Completed, JSR at FC49/FC87 in Move Next Byte from Save Area and Increment Pointer.

The pointer to the current byte being loaded or saved, (AC), is incremented.

### **Operation:**

1. INC AC, the low pointer to the current byte being saved/loaded.
2. If AC is not equal to zero after the increment then no page boundary was crossed, so branch to step 4 to RTS.

3. If AC is zero then a page boundary was crossed so increment AD the high pointer to the current byte being saved/loaded.
4. RTS.

### **Determine If Open Is for Read or Write F38B/F44B-F398/F458**

#### **Called by:**

BNE at F386/F446 in OPEN Execution.

First, check the tape buffer address in (B2) to make sure the tape buffer does not start below 0200. If it does start below 0200, then display Illegal Device Number message and RTS.

Next, determine whether the secondary address B9 indicates a read (secondary address of 0) or a write (nonzero secondary address) to tape.

Branch to F3B8/F478 if writing to tape. Fall through to F399/F459 if reading from tape.

#### **Operation:**

1. JSR F7D0/F84D to determine if tape buffer address (B2) is greater than or equal to 0200. If not, then JMP F713/F796 to display Illegal Device Number message if kernal control messages flag is on, then RTS from this routine.
2. LDA B9, the current secondary address.
3. Mask off the high nybble as the Open since serial routines have placed a \$6 in the high nybble.
4. If the value remaining in the accumulator is nonzero, then we're doing an open for tape write; branch to F3B8/F478
5. If the value remaining in the accumulator is zero, then we're doing an open for tape read; fall through to F399/F459.

### **Open Logical File for Reading from Tape F399/F459-F3D4/F494**

#### **Called by:**

Falls through from F398/F458 in Determine If Open Is for Read or Write.

Display the PRESS PLAY ON TAPE message if no tape buttons are down. If the keyboard STOP key is down then exit.

Display the SEARCHING. If a filename has been specified, then also display the filename.

## Tape I/O Routines

If no characters are in the filename, then just load the next header.

If the filename does contain characters, then load the specified filename.

On return from loading the header if an end-of-tape header (tape identifier of 5) was loaded, then RTS with the accumulator containing 5 and carry set, allowing BASIC to display the DEVICE NOT PRESENT message.

If the STOP key was pressed, then display FILE NOT FOUND error message if using machine language, returning with 4 in the accumulator and carry set. BASIC displays the error message ?BREAK if the STOP key is pressed.

If indeed a header with an identifier of 1, 3, or 4 was found, then set A6 to 191 characters in the tape buffer and RTS. Setting A6 to 191 forces the first CHRIN for this tape file to load the buffer again with the first 192 byte data area following on tape.

It appears that tape identifiers of 1 or 3, normally associated with program tapes, can function the same as tape headers of 4 when loading a sequential file. Of course, when writing a sequential file, a tape identifier of 4 is used for the header.

### Entry conditions:

B9 contains current secondary address with low nibble = 0 for a tape read. B7 contains number of characters in filename.

### Operation:

1. JSR F817/F894 to see if any tape buttons are down. If none are down, then display PRESS PLAY ON TAPE and then display OK when one is pressed. If the keyboard STOP key is pressed then BCS to F3D4/F494 to RTS.
2. JSR F5AF/F647 to display SEARCHING. If a filename is being used, then also display FOR and the filename.
3. LDA B7, the number of characters in the filename.
4. If no characters in the filename then branch to step 9.
5. If characters in the filename, then JSR F7EA/F867 to load the header containing this filename into the tape buffer.
6. BCC to step 13. Carry is clear on return from step 5 if header with tape identifier of 1, 3, or 4 is found that contains the filename specified.
7. BEQ to step 17 to RTS. Z = 1 (BEQ condition) if header with tape identifier of 5 indicating end-of-tape was loaded.

- RTS with accumulator holding 5 and carry set. (Carry is set on a compare that is equal).
8. If carry set, but  $Z = 0$ , then the keyboard STOP key was pressed and detected. JMP F704/F787 to set error number of 4, display the message FILE NOT FOUND, and close any open files. Since BASIC uses its own messages, this FILE NOT FOUND message is not displayed from BASIC. Rather, ?BREAK ERROR is displayed.
  9. Branch here from step 4 if no filename is specified. In this situation just load the next header by JSR F72C/F7AF.
  10. BEQ to step 17 to RTS.  $Z = 1$  (BEQ condition) if header with tape identifier of 5 indicating end-of-tape was loaded. RTS with accumulator holding 5 and carry set. (Carry is set on a compare that is equal).
  11. BCC to step 13. Carry is clear on return from step 5 if header with tape identifier of 1, 3, or 4 is found.
  12. If carry set, but  $Z = 0$ , then the keyboard STOP key was pressed and detected. Branch to step 8.
  13. F3C2/F482: LDA \$BF (decimal 191).
  14. See if secondary address indicates a read, which it should. Branch to F3D1/F491, step 15.
  15. STA A6, setting it to 191.
  16. CLC.
  17. RTS.

### **CHRIN from Tape F179/F230-F198/F24F**

#### **Called by:**

Fall through from F178/F22F in Determine Input Device.

Return the next byte from the tape buffer in the accumulator. Also, read one byte ahead to see if the next byte is zero, indicating end-of-file, and if true then set end-of-file status in 90.

Since the X register is saved at entry to this routine, and restored at exit, it can be used as an index register in the routine that calls CHRIN.

#### **Operation:**

1. STX in 97 to allow X register to be restored to its entry value upon exit from routine.
2. JSR F199/F250 to increment the pointer to the tape buffer A6. If A6 contains 192 after the increment, then load the

## Tape I/O Routines

---

tape buffer from tape again, and reset A6 to 0.

If A6 was reset to zero, then the A6 is again incremented (resulting in a value of 1). Thus, the tape identifier is skipped during CHRIN.

Return the byte pointed to by A6 after it was incremented.

If the keyboard STOP key was pressed, then branch to step 5 to restore X register from 97, then RTS.

3. PHA, storing the byte just returned from the tape buffer.
4. Now read one byte ahead in the tape buffer to determine if end-of-file status in 90 should be set. End-of-file is set if the next byte is a zero. You can check the end-of-file status from your BASIC or machine language program.

To read one byte ahead, again JSR F199/F250, returning the next byte from the tape buffer in the accumulator.

If the keyboard STOP key was pressed, then branch to F193/F24A to PLA to keep the stack in proper order, then restore X register from 97 and RTS.

If the byte from the read ahead is 0, then JSR FE1C/FE6A to set end of file status in 90 of \$40.

Finally, decrement A6, as it was incremented in this one byte look-ahead step.

5. Restore the X register that was saved in 97.
6. PLA to restore from the stack the byte returned from the tape buffer.
7. RTS.

### **Return Byte from Tape Buffer F199/F250-F1AC/F263**

#### **Called by:**

JSRs at F17B/F232 and F181/F238 in CHRIN from Tape.

Increment the counter/pointer to the tape buffer, A6.

If A6 is not 192 then no need to read another tape buffer, just return the byte from the tape buffer pointed to by A6.

If A6 is 192, then load the next tape buffer. Reset A6 to 0, then branch to the start of this routine again, thus incrementing A6 past the tape identifier to return the second byte from the tape buffer.

No check appears to be made when loading the next tape buffer that the buffer contains a tape identifier of 2 indicating a data buffer. Rather, the next 2 blocks are loaded. However, it



does appear that long block or short block status will be set if a program file is loaded as if it were a data file, and your program could check the status for the long or short block status. Long block status should be set if the program is longer than 192 bytes long; if program is under 192 bytes long then short block status should be set.

### **Operation:**

1. JSR F80D/F88A to Increment Count of Characters In Tape Buffer. Also compare this pointer to 192 to see if at the end of the buffer.
2. If A6 is not equal to 192 then branch to step 7 to return the next byte from the tape buffer.
3. If A6 is equal to 192, then JSR F841/F8C0 to load the next header into the tape buffer.
4. If the keyboard STOP key is pressed, then branch to step 9 to RTS.
5. Reset A6 to zero.
6. Branch to step 1. Thus, upon loading a new tape buffer, A6 is incremented to a value of 1, bypassing the tape identifier.
7. LDA (B2),Y. Load accumulator with the next byte from the tape buffer. The Y register is set from value in A6 after A6 was incremented.
8. CLC.
9. RTS.

### **Control Routine for Tape Load F539/F5D1-F5AE/F646**

#### **Called by:**

BCS/BNE at F534/F5CC in Determine Device for LOAD.

If the tape buffer address is greater than or equal to 0200, then load the tape buffer with a header from the tape, either a specific one if a filename is given or the next one if no filename is used. However, only tape headers with tape identifiers of 1 or 3 are acceptable for use. A tape identifier byte of 5 indicates end-of-tape header in which case this routine just exits with the accumulator containing 5 and the carry set. If a tape identifier of 2 or 4 is read, the tape header is not used as these tape identifiers are used for sequential files; search for the next tape header with tape identifier of 1, 3, or 5 in this case.

After loading the header, the tape identifier is examined. A tape identifier of 3 causes a nonrelocatable load during which the starting address is taken from the second and third bytes of the tape buffer. A tape identifier of 1, along with a secondary address of 0, allows a relocatable load with the values in the X and Y registers at entry to the load used as the starting address for the load. However, a tape identifier of 1 and a nonzero secondary address force a nonrelocatable load.

The length of the program to be loaded is determined by subtracting the starting address in the tape buffer from the ending address in the tape buffer. This length is then added to the actual starting address to obtain the ending address of where to load the program. The actual starting address is taken from the starting address in the tape buffer unless a relocatable load is specified.

Then, the LOADING or VERIFYING message is displayed, and the next two program blocks are loaded into the load area.

Finally, at exit from the program load, (AE) contains the address of the end of the load.

While an end-of-tape header is detected and returned to BASIC as the value of 5 in the accumulator and the carry set, allowing BASIC to display the message DEVICE NOT PRESENT error message, the Kernal routine for load does not explicitly set 90, the status byte, to indicate this error. This is contrary to the table displayed for READST in the 64 and VIC *Programmer's Reference Guides*. Other tape error conditions of short block, long block, unrecoverable read error, and checksum are flagged in 90. To test for end-of-file from a machine language program, it appears you have to examine the carry status and/or the accumulator on return from the JSR FFC0 for load. The carry is set and accumulator contains 5 if end-of-tape. Also, carry is set and accumulator contains 4 if the STOP key was pressed. READST (JSR FFB7) does not return the status for end-of-tape following an load.

When loading a program tape, either the program is loaded back into the same area it was saved from or it is relocated to a different area of memory. Two factors determine whether a tape will be loaded back into the same place it was saved from. First, however, let's define the terms used here. A relocatable program tape is one that has been saved in such a manner that it is possible (though not required) to load it into a different area of memory than the area from which it was

saved. A nonrelocatable program tape only allows the program to be loaded back into exactly the same memory locations from which it was saved. The first factor that determines whether a program tape is relocatable or nonrelocatable is the tape identifier that is the very first byte of data in the tape header. If this identifier is 3, then the program tape is nonrelocatable. If the identifier is 1, then the program tape is relocatable. However, the second factor of the secondary address in B9 enters into consideration with a program tape whose header tape identifier is 1. Although a program with a tape identifier of 1 is relocatable, it is only relocated if you use a secondary address of zero. The area to which it is relocated is specified in the X and Y registers before JSR FFD5 to the kernal routine load. A nonzero secondary address causes a program tape with tape identifier of 1 to be nonrelocatable.

When a tape is nonrelocatable, it retrieves the location where the tape is to be loaded from bytes 2 and 3 of the tape buffer, which contain the starting address from which the program tape was saved.

A tape identifier of 1 is written on a tape during tape save when an even secondary address is specified. A tape identifier of 3 is written during tape save when an odd secondary address is specified.

Because the tape load operation is really a two-part step in which the header is loaded and then the program, you could execute each step individually and modify the starting address and ending address for the program after the header has been loaded.

### **Entry conditions:**

B9 contains the secondary address. (C3) contains the starting address for save from the X and Y registers if this is a relocatable load. (B2) contains the address of the tape buffer.

### **Exit conditions:**

(AE) contains ending address of program loaded.

### **Operation:**

1. JSR F7D0/F84D to see if the tape buffer starts greater than or equal to 0200. If it does not, then exit with ILLEGAL DEVICE error message.
2. JSR F817/F894 to see if any tape buttons are down. If none are down, then display the PRESS PLAY ON TAPE

message. Test for a button depressed and display OK when one is detected down. If the keyboard STOP key was pressed then RTS.

3. JSR F5AF/F647 to display SEARCHING and if a filename was specified then also display FOR and the filename.
4. A loop is now performed to look for the next tape header until a header with a tape identifier of 1 or 3 is found.  
First, LDA B7 to see if there are any characters in the filename. If there aren't, then branch to step 9.
5. If there are characters in the filename, then JSR F7EA/F867 to load the header containing this filename into the tape buffer.
6. BCC to step 13. Carry is clear on return from step 5 if header with tape identifier of 1, 3, or 4 is found that contains the filename specified.
7. BEQ to RTS.  $Z = 1$  (BEQ condition) if header with tape identifier of 5 indicating end-of-tape was loaded. RTS with accumulator holding 5 and carry set.
8. If carry set, but  $Z = 0$ , then the keyboard STOP key was pressed and detected. BCS F530/F5C7 which JMPs F704/F787 to set error number of 4, display message FILE NOT FOUND, and close any open files. Since BASIC uses its own messages, this FILE NOT FOUND message is not displayed from BASIC. Rather, ?BREAK ERROR is displayed.
9. Branch here from step 4 if no filename is specified. In this situation just load the next header by JSR F72C/F7AF.
10. BEQ to RTS.  $Z = 1$  (BEQ condition) if header with tape identifier of 5 indicating end-of-tape was loaded. RTS with accumulator holding 5 and carry set.
11. If carry set, but  $Z = 0$ , then the keyboard STOP key was pressed and detected. BCS F530/FC57 which JMPs to F704/F787 to set error number of 4, display message FILE NOT FOUND, and close any open files.
12. If carry is clear on return from step 9, then header with tape identifier of 1, 3, or 4 is found.
13. Before proceeding further, check 90 to see if an unrecoverable read error occurred during load of the header. If so, then branch to step 32 to RTS.
14. Since only program headers with tape identifiers of 1 or 3 are wanted, tape identifiers of 4 will be rejected and a search for the next header is performed.

## Tape I/O Routines

---

15. CPX \$01. The X register contains the tape identifier byte. If this compare is equal, then a tape identifier of 1 for a relocatable program tape has been found. Branch to step 21.
16. CPX \$03. If not equal (if a tape identifier byte of 4 for example), then branch to step 4 to load another header.
17. If the X register contains 3, then a tape identifier byte for a nonrelocatable program file has been loaded; continue with step 18.
18. Load the second byte in the tape buffer (the low byte of the address of the program area that was saved) and store the value in C3.
19. Load the third byte in the tape buffer (the high byte of the address of the program area that was saved) and store the value in C4.  
(C3) now contains the starting address of the program that was saved to tape.
20. BCS to step 23. When a comparison is equal, such as  $X = 3$  or  $X = 1$  then the carry status is set. Thus, this branch is unconditional.
21. Branch to this step from step 15 if a tape identifier of 1 was found. LDA B9, the secondary address.
22. If the secondary address is nonzero, then a nonrelocatable load has been specified, so branch to step 18. If a zero secondary address has been specified, then a relocatable load is requested. The values in the X and Y registers at entry to the load are stored in (C3).
23. Now compute the length of the program to be loaded.  
LDA with the fourth byte from the tape buffer, which is the low byte of the address of the end of the program that was saved. Subtract the second byte in the tape buffer from the accumulator; the second byte is the low byte of the address of the start of the program that was saved.  
Transfer the results to the X register for temporary storage.
24. LDA with the fifth byte from the tape buffer, which is the high byte of address of the end of the program that was saved.  
Subtract the third byte in the tape buffer from the accumulator; the third byte is the high byte of the address of the start of the program that was saved.  
Transfer the results to Y register for temporary storage.

25. Now that the length is known, add the length to the starting address (either relocatable from X and Y registers or nonrelocatable from the second and third bytes in tape buffer) to obtain the ending address for the load.
26. Transfer X register back to accumulator. Add C3, the low byte of the address of the start of the program, either from the tape buffer or if relocatable then from X register value at entry to load. Store the result in AE, the low byte of the address of the end of the program.
27. Transfer the contents of the Y register back to the accumulator. Add C4, the high byte of the address of the start of the program, either from the tape buffer or if relocatable then from the Y register value at entry to load. Store the result in AF, the high byte of the address of the end of the program.
28. Set (C1) from (C3).
29. JSR F5D2/F66A to display either LOADING or VERIFYING.
30. JSR F84A/F8C9 to load the program from the two program blocks following the header.
31. At completion of program load, load the X register from AE, the low byte of the address of the end of the program loaded, and load the Y register from AF, the high byte of the address of the end of the program loaded.
32. RTS.

### **Find Specified Tape Header F7EA/F867-F80C/F889**

#### **Called by:**

JSR at F3A5/F465 in Open Logical File for Reading from Tape, JSR at F54D/F5E5 in Control Routine for Tape Load.

Load headers (tape identifiers of 1, 3, or 4) until a header is found whose name matches the filename specified in the OPEN or LOAD. Stop the search when an end-of-tape header is found or if the keyboard STOP key is pressed and detected.

While the maximum length of a filename in a tape buffer or a tape header is 187 bytes, this routine to find a specified header allows a filename of up to 256 bytes to be compared to the filename in the tape buffer. However, it should always find a mismatch if a filename over 187 bytes is specified unless you somehow filled the area past the tape buffer with bytes equal to the characters in the filename past position 187.

### Exit conditions:

Carry set if STOP key pressed or if an end-of-tape header with matching filename was loaded (also, if end-of-tape found the Z = 1 flag of status register (BEQ condition) is set).

Carry clear if the specified header is found.

If a header is found that matches the filename specified, then the tape buffer contains this header.

### Operation:

1. JSR F72C/F7AF to load the next tape header with a tape identifier of 1, 3, 4, or 5 into the tape buffer.
2. If a tape header with a tape identifier of 5 indicating an end-of-file header is loaded, then branch to step 12 to RTS with the accumulator holding 5 and the carry set. Also, if the STOP key was pressed then BCS to step 12.
3. Set 9F, the index into the tape buffer, to 5.
4. Set 9E, the index into the filename, to 0.
5. Compare 9E to B7, the number of characters in the filename. If equal then branch to step 11 to CLC and RTS.
6. Load the next character from the filename.
7. Compare this character to the next character from the tape buffer.
8. If not equal, then branch to step 1 to get the next tape header.
9. If equal, then increment 9E and 9F.
10. If 9E is not \$00 then branch to step 5. If 9E = \$00 then fall through to step 11. However, it is unlikely 9E will roll over to zero. Since this routine was executed because B7 was nonzero, then 9E should eventually match B7.
11. CLC.
12. RTS.

### Find Next Tape Header F72C/F7AF-F769/F7E6

#### Called by:

JSR at F3AF/F46F in Open Logical File for Reading from Tape,  
JSR at F556/F5EE in Control Routine for Tape LOAD, JSR at  
F7EA/F867 in Find Specified Tape Header.

Load the next two tape blocks into the tape buffer.

If the keyboard STOP key was pressed, then exit with carry set.

If the first byte in the tape buffer is 5, indicating an end-of-file header was loaded, then exit with accumulator holding 5 and carry set.

If the first byte in the tape buffer is 1, 3, or 4 then consider this buffer an acceptable tape header. Display FOUND and the filename from the tape buffer, then exit.

If the first byte in the tape buffer is not 1, 3, 4, or 5 then repeat this entire sequence of searching for the next header.

### **Exit conditions:**

Carry set if keyboard STOP key was pressed and detected, or if end-of-file header was loaded. (Also, if end-of-file header was loaded, then  $Z = 1$ , the BEQ condition).

Carry clear if header with tape identifier of 1,3, or 4 was found.

The tape buffer is loaded with the header from tape.

### **Operation:**

1. Save the LOAD/VERIFY flag, 93, on the stack.
2. JSR F841/F8C0 to read the next two blocks on tape into the tape buffer.
3. Restore 93 from the stack.
4. If STOP key was pressed and detected, then BCS to step 15 to RTS with accumulator holding 0 or 1.
5. Load the first byte of the tape buffer.
6. If this first byte is 5, then an end-of-tape header was just loaded. In this case branch to step 15 to RTS with accumulator containing 5 at exit and carry set.
7. If first byte in tape buffer is 1, 3, or 4, then continue with step 8.
8. Branch to step 1 if first byte of tape buffer is not 1, 3, 4, or 5.
9. TAX, transferring the first byte from the tape buffer to X register.
10. If 9D, the Kernal message control, has its high bit off, branch to step 13.
11. JSR F12F/F1E6 to display the FOUND message.
12. Starting from the tape buffer + 5, at the start of the filename in the tape buffer, output the next twenty characters from the tape buffer to the output device, typically the screen.
13. CLC.



14. DEY. Thus Y will hold 19 (hex \$14) on exit. I do not see the purpose of this instruction.
15. RTS.

### **Read Tape Header into Buffer F841/F8C0-F849/F8C8**

#### **Called by:**

JSR at F19E/F255 in CHRIN from Tape, JSR at F72F/F7B2 in Find Next Tape Header. This is the entry point for reading a tape header (reading whatever next two blocks occur into tape buffer).

JSR F7D7/F854 to set start and end of tape buffer, setting the starting and ending addresses for the load. The starting address for the load (C1) is set from (B2) the tape buffer pointer. The ending address for the load (AE) is the starting address + 192.

Fall through to F84A/F8C9 to load the next two blocks into the load area.

#### **Operation:**

1. Set 90, status byte to 0. Also set 93, the LOAD/VERIFY flag to 0 indicating a load.
2. JSR F7D7/F854 to use tape buffer address as starting address for load (C1), and tape buffer address + 192 as ending address for load (AE).
3. Fall through to F84A/F8C9 to load the next two blocks from tape into the tape buffer.

### **Load Next Two Blocks F84A/F8C9-F863/F8E2**

#### **Called by:**

Falls through from F847/F8C6 in Read Tape Header into Buffer, JSR at F5A2/F63D in Control Routine for Tape Load.

Prepare variables and counters for tape load.

Prepare X register and accumulator for changing IRQ vector to F92C/F98E for tape read and for enabling FLAG/CA1 interrupts.

Branch to F875/F8F4 to make this vector change and enable the FLAG/CA1 interrupt.

### Operation:

1. JSR F817/F894 to see if any tape buttons are down. If none are down then display the PRESS PLAY ON TAPE. Then check that a button is down and display OK. If the keyboard STOP key is pressed and detected, branch to set 02A0 to 0, then RTS.
2. Initialize counters and variables as follows:  
AA = 0, B4 = 0, B0 = 0, 9E = 0, 9F = 0, 9C = 0.
3. LDA \$90/\$82 in preparation for setting the CIA #1 interrupt enable register/VIA #2 interrupt enable register to enable FLAG/CA1 interrupts for tape read.
4. LDX \$0E to prepare to index into table of IRQ vectors to retrieve the vector F92C/F98E for reading from tape.
5. Branch to F875/F8F4 to change the IRQ vector, enable FLAG/CA1 interrupt, and make other preparations for the tape load.

### Tape Read Overview

Whenever a FLAG/CA1 Interrupt occurs for the tape read routines, then the IRQ interrupt service routine at F92C/F98E is executed. However, this IRQ interrupt service routine can be thought of as two separate sections. In one section at F92C/F98E–FA5F/FAAC the individual bits are read from the tape. Once this section has found a leader and the following word marker and thus prepared the various flags for receiving data bytes, another section comes into consideration. This other section at FA60/FAAD–FB67/FBAB is passed a byte when the first routine concludes it has received enough bits to make a byte (eight data bits and a parity bit). The second section then determines what to do with the byte it just received—for example, whether to treat it as a block countdown character or as a valid data byte to be loaded into the load area.

No one method seemed adequate for describing these two sections as the flow of control in each section is by no means straightforward.

I ended up describing these two sections in this manner: instructions are grouped together by function with each function labeled. These groups are listed sequentially if that seemed reasonable. The *Called by* section will list the reasons this routine is called. While these groups are not separate sub-

routines that can be branched to and returned from by an RTS, the groups do seem to serve separate functions.

If you are not already familiar with the tape routines, it will probably take several readings of these two sections to understand how the tape read works, as tape read seems much more complex than the tape write routines.

A topic that deserves treatment is the speed as which tapes are saved and loaded on the 64/VIC and the reliability of the saves and loads.

The following calculation shows a maximum rate of writing to tape for the VIC. A bit is written to tape in 345.8 microseconds for a 0 dipole and 502.4 microseconds for a 1 dipole, or a total of 848.2 microseconds. One second divided by 848.2 microseconds/bit = 1179 bits/second maximum recording speed to cassette. However, due to the leaders, the recording of two blocks, block control characters, etc., the actual number of bits/second recording speed is much less than this maximum.

Commodore has left the tape save and load routines virtually unchanged from the PET to the VIC to the Commodore 64. The reliability of the tape routines is excellent, as there are two levels of error checking on the load: parity for each byte and parity (or checksum) over all bytes loaded. Along with the parity check for each byte is the possibility of correcting errors in the first block by the corresponding byte from the second block.

Along with this reliability of the tape routines comes the fact that by keeping the tape routines the same tapes can be loaded on any of the Commodore computers. Thus, even if a faster or more reliable set of tape save/load routines is available, switching to a different set of tape save/load routines would introduce the problem of compatibility of the old format tapes with the new format tape save/load routines. Unless the new tape load/save routines could routinely handle the old format tapes, either both routines would have to be provided or the old format tapes could no longer be loaded.

The only obvious place to speed up the tape load routine appears to be the section at the end of loading the first block. If no errors are observed during the first block load, then the checksum could be computed at this point. If the computed checksum matches the checksum from the save, then the load could be considered complete, although this would leave the tape physically positioned at the tape head between the first

and second blocks that were saved for the program. This positioning of the tape would make such a modification incompatible with sequential files, rather the modification could only be used when loading program files.

Programs are available for the 64/VIC that provide faster tape routines, such as the Rabbit and the Arrow. Without examining the actual code in the Rabbit, just listening to a tape recorded in Rabbit format on a standard audio cassette it sounded to me as if only one block is used rather than two, that the leader time is much shorter, and that the time per bit is shorter. I did not have time to see how the Rabbit actually worked. The Rabbit does support both normal format VIC tapes and the fast Rabbit tapes. If you are not concerned about exchanging tapes with other users and only concerned about speeding up the save/load routines for your own programs or sequential files, either the Rabbit or the Arrow might be a worthwhile investment if you use tape a lot and don't have a disk drive.

### **Determine Time Between FLAG/CA1 Interrupts for This Dipole F92C/F98E-F939/F99B**

#### **Called by:**

Occurrence of IRQ interrupt (normally a FLAG/CA1 interrupt, but a timer A/timer 1 interrupt is also possible).

The time between FLAG/CA1 interrupt occurrences is determined for this dipole. This dipole time is later used to determine whether the dipole just read is to be considered noise, 0, 1, or a word marker.

What is a dipole? See Figure 10-2 for a pictorial representation of a dipole.

The positive and negative voltages that are recorded onto the tape result in two different poles—just like a magnet with a north pole and a south pole. This recording onto the tape can be represented as a square wave as shown in Figure 10-1. A dipole (literally two poles) time is the time for two poles (two square wave cycles).

Two dipoles make up a bit. Leader dipoles are all of a short-0 time length, two 0 cycles. Data bits are either a short-0 dipole followed by a medium-1 dipole, representing a 0 data bit, or a medium-1 dipole followed by a short-0 dipole,

representing a 1 data bit. Word marker bits (used to signify the end of the leader cycles and the end of each byte) are a long-word marker dipole followed by a medium-1 dipole. I use the term word marker both in referring to a data bit and to the dipole that makes up the first half of this data bit. Whether I am referring to the dipole or the bit is hopefully obvious from the context.

Short-0, Medium-1, and Long-Word Marker refer to the relative times for the dipoles; these times can vary slightly. See the description of the variable 92 for some representative values with which the tape routines start.

I sometimes interchange the way I refer to these dipoles, that is sometimes I use short, other times 0. Similarly, at times medium is used, while other times 1. The same interchange applies to long-word marker.

The BNE F92C/F98E at F938/F99A limits the maximum timer B/timer 2 read error to 14 cycles rather than a possible 256 if this BNE was not here.

Figure 10-17 displays the number of cycles each instruction in this routine takes on the VIC.

**Figure 10-17. Tape Read: Limiting Timer B/Timer 2 Error**

Location	Instruction	Cycles
F98E	LDX 9129	4
F991	LDY \$FF	2
F993	TYA	2
F994	SBC 9128	4
F997	CPX 9129	<u>4</u>
		16 cycles (through F997)
F99A	BNE F98E	

Now if the high byte of timer 2 in 9129 has changed at F997 from its value at entry point F98E then at F98E the low byte of timer 2 was within 16 cycles of decrementing through zero, and thus decrementing the high byte of the counter.

For example, if the first time through this routine at F98E timer 2 has a value of \$2504, then at F997 timer 2 will be \$2504 - \$10 (decimal 16), or \$24F4. The CPX at F997 would find X register of \$25 but 9129 of \$24, and thus branch to F98E to read the timer again, and thus introduce an error into the timer 2 reading of 16 cycles. Since the two low bits of the

timer are thrown away anyway, this is not enough error to affect the tape read routines in determining whether the dipole just read was noise, 0, 1, or word marker.

However, consider what would happen without this BNE. At exit from F997 the X register would contain \$25 rather than the value of \$24. Since this is the high byte of the timer count, and a 1 is equivalent to 256 cycles of the timer, the value in the X register could be up to 256 cycles in error. This error in the X register would be large enough to affect the readings for the dipole.

On the 64, timer B is set to run in one-shot mode. In one-shot mode the timer counts down to zero, generates an interrupt, reloads the latched value, and stops. However, it seems that the interrupt would have to be serviced before the latched value is loaded into timer B and timer B stops or else this routine would be reading the latched value in the count and not the count that reflects the timer B was started. The 6526 data sheet does not define the one-shot mode in enough detail to know if this is what is actually done.

### **Entry conditions:**

IRQ interrupt has occurred.

### **Exit conditions:**

The X register contains high byte of timer B/timer 2 count. The accumulator contains \$FF — low byte of timer B/timer 2 count. The Y register contains \$FF.

### **Operation:**

1. Load X register with the high byte of the timer B/timer 2 count. Timer 2 decrements continuously, and when reaching zero, just rolls over and starts counting again from \$FFFF. Timer B is running in one-shot mode, but see the previous comments.
2. Load Y register with \$FF.
3. TYA.
4. SBC DC06/9128. This SBC (which clears the timer 2 interrupt flag on the VIC) leaves in the accumulator the low byte of the timer count since timer B/timer 2 was last set to \$FFFF. Timer B/timer 2 was last set to \$FFFF during the routine immediately following this one, at F940/F9A2.
5. CPX DC07/9129. See if the high byte of timer B/timer 2 has changed from its value at entry step 1. If so, then the

low byte of timer B/timer 2 was within 16 cycles of rollover.

6. If the compare is not equal in step 5, indicating the high counter has changed, then branch to step 1 to read timer B/timer 2 again, thereby preventing the large timer B/timer 2 error that could affect the accuracy of dipole readings.

## **Convert Time Between Interrupts into One-Byte Value F93A/F99C-F95A/F9AF**

### **Called by:**

Falls through from F939/F99B whenever an IRQ interrupt occurs.

The time since timer B/timer 2 was set \$FFFF minus the value of timer B/timer 2 read in the previous routine is converted into a one-byte value in B1 by dropping the two low bits and the six high bits from this value of \$FFFF — timer B/timer 2.

This value in timer B/timer 2 that is converted to the one-byte value in B1 is approximately equal to the time value of a dipole. Indeed, if you add the 20 cycles (20.4 microseconds) between the read of timer B/timer 2 to the reset of timer B/timer 2 to \$FFFF to this value of (\$FFFF — timer B/timer 2), you obtain the exact time for the dipole just read.

### **Entry conditions:**

The X register contains the high byte of timer B/timer 2 count. The accumulator contains \$FF — low byte of timer B/timer 2 count. The Y register contains \$FF.

### **Exit conditions:**

B1 contains value representing in one byte the two-byte value of \$FFFF — timer B/timer 2 read in previous routine.

### **Operation:**

1. STX B1, saving the value of the high byte of the timer B/timer 2 high count read in previous routine.
2. TAX. Transfer the accumulator, which contains \$FF — low byte of timer B/timer 2 count, to the X register for temporary storage.
3. Reset timer B latches/timer 2 counters to \$FFFF by storing the Y register containing \$FF into DC06/9128 and DC07/9129. For the VIC, storing the value in 9129, the high byte

of the timer count, clears the timer 2 interrupt flag, transfers the latched value for the low byte to the low byte of the counter, and starts timer 2 counting down from \$FFFF. On the 64, it is necessary to store \$19 in DC0F to load the timer B latches into the timer B counter, set one-shot run mode, and start timer B. Also LDA DC0D to clear any timer B interrupt flag.

4. TYA. Accumulator now contains \$FF.
5. Subtract B1; B1 contains the high byte of the counter value for timer B/timer 2.
6. STX B1. Put \$FF — low byte of timer B/timer 2 count into B1.  
As a result, B1 now contains \$FF — low byte of timer B/timer 2 count, and the accumulator contains \$FF — high byte of timer B/timer 2 count.
7. Now the two low bits of B1 are dropped and the two low bits of accumulator shifted into bits 7-6 of B1, as illustrated in Figure 10-18 (each dot represents a bit).

**Figure 10-18. Converting Two-Byte Timer Value to One-Byte Indicator**

	Accumulator	Carry	B1
	A7 ..... A0	•	B7 ..... B0
LSR	• A7 ..... A1	A0	
ROR \$B1		B0	A0 B7 ..... B1
LSR	•• A7 ..... A2	A1	
ROR \$B1		B1	A1 A0 B7 ..... B2

Thus, at exit from this series of shifts and rotates B1 contains bits 1 and 0 of the accumulator and bits 7-2 of the starting value of B1.

What values can B1 represent now?

Figure 10-19 pictures timer B/timer 2 and the bits used from it, along with the maximum value:

**Figure 10-19. Maximum Value for B1**

High byte						Low Byte										
x	x	x	x	x	x	T9	T8	T7	T6	T5	T4	T3	T2	x	x	
0	0	0	0	0	0	1	1	1	1	1	1	1	1	0	0	= \$03FC (1020 decimal)

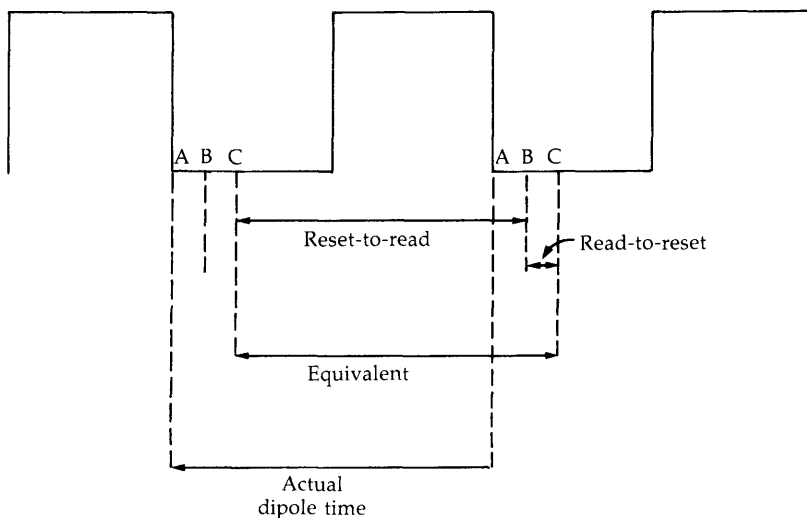


Timer 2 of 1022 ( $1020 + 2$  cycles for reset when it hits 0) divided by 1,022,370 is equal to 999.6 microseconds maximum value that timer 2 can represent once it is shifted into the one-byte B1. This time is adequate to frame any 0, 1, or word marker dipole. However if a dipole went above 999 microseconds, then the timer B/timer 2 reading for it would not be correct as only up to 999 microseconds can be uniquely determined.

When doing a tape load, you can stop the tape and examine B1 to see the value obtained in reading the last dipole. This value just represents the time between the last setting of timer B/timer 2 to \$FFFF and the read of timer B/timer 2. To obtain the total time for the dipole, the time between read of timer B/timer 2 and reset to \$FFFF of 20.4 microseconds must be added to the time that B1 represents. B1 can be converted to a time value by shifting it left two bits (or multiplying by 4) and then dividing this value by 1,022,370.

Figure 10-20 illustrates why it is necessary to add the timer B/timer 2 read to timer B/timer 2 reset time to the time represented by B1 to obtain a value for the dipole time:

**Figure 10-20. Reading Dipoles**



- A FLAG/CA1 interrupt
- B Timer B/2 read
- C Timer B/2 reset

## Tape I/O Routines

The read-to-reset time consists of the instructions (on the VIC starting at F997 and ending at F9A2) of:

Location	Instruction	Cycles
F997	CPX 9129	4
F99A	BNE F98E	3
F99C	STX B1	3
F99E	TAX	2
F99F	STY 9128	4
F9A2	STY 9129	4
		<hr/> 20 cycles

The 20 cycles total  $\times$  1.022370 microseconds/cycle gives 20.4 microseconds between read of timer 2 and reset of timer 2.

The tape read routines read dipoles while the tape write routines write cycles (or half dipoles).

### Determine If Dipole Time Represents Noise, 0, 1, or Word Marker

**F959/F9B0-F98A/F9E4**

#### Called by:

Falls through from F958/F9AF, Convert Time Between FLAG/CA1 Interrupts into One-Byte Value.

Determine if the dipole time just read is noise, 0, 1, or a word marker. Clear the interrupt flag set by the cassette read. Once the first dipole of a word marker bit (word marker dipole) signifies end of byte and sets 9C to nonzero, the second dipole of the word marker bit (1 dipole) causes a jump to FA60/FAAD to handle the byte just received.

If a bit is received that has a dipole time for a word marker dipole, then JMP FA10/FA60. Normally, this dipole is the one following a group of leader bits. Also, JMP FA10/FA60 if A3 has its high bit on, indicating the previous parity bit has been received. This jump if the parity bit has been received usually occurs for the read of the first dipole of the word marker bit at the end of a byte. Thus, the dipole following a parity bit is not checked for the various time ranges (except to check that it is longer than noise).

#### Entry conditions:

B0 contains a value (initially zero) used in setting time limits, referred to as adjustable baselines for what are considered noise, 0, 1, and word marker dipoles. A B0 value below 0

(255, 254, 253, etc. in decimal) decreases the adjustable baseline time, while a B0 value above 0 (1, 2, 3, etc.) increases the adjustable baseline times. B1 contains a value corresponding to the time of the dipole just read from tape (minus the time between reading timer B/timer 2 and resetting timer B/timer 2, but this read-reset time is small enough that it can be ignored in the comparisons). A3, the count of bits remaining to be read for a bit, is initialized to 8 before each byte, and decremented after each bit is read. When decremented to -1, then the parity bit has just been read. 9C contains 0 when waiting for the next byte or when receiving current byte, or 1 when a byte has been completely received (but before it is processed by the byte handler routine).

## Exit conditions:

The interrupt flag for tape read is cleared.

## Operation:

1. LDA B0, which is used to adjust baselines.

CLC

ADC \$3C. Add time for setting limit of what is considered noise, the noise/0 adjustable baseline.

CMP B1. Compare this time limit for noise to the current time for this dipole.

BCS F9AC/FA06 If the time for current dipole is less than the noise/0 adjustable baseline, then it must be noise. At F9AC/FA06 usually branch to F9D2/FA22 to JMP to FEBC/FF56 to restore registers and RTI.

Let's look at some values that are placed in the accumulator as the result of the LDA B0, CLC, ADC\$3A for various values of B0.

Values of B0 and corresponding value in accumulator (converted to a time value by shifting left two bits and dividing by 1,022,370) that represents the noise/0 adjustable baseline are shown in Figure 10-21.

**Figure 10-21. Noise/0 Adjustable Baseline Times**

	B0: 252	253	254	255	0	1	2
Noise/0 adjustable baseline time: (in microseconds)	224	228	232	236	240	244	248

2. LDX 9C. During tape load, 9C is 0 when waiting for the next byte or when receiving the current byte, 9C is 1 when

a byte (8 data bits and parity bit) have been completely received.

BEQ F9C9/F9C3. Branch to step 3 if 9C is 0, indicating that a byte has not yet been completely received.

If 9C is nonzero, then a byte has been completely received. The word marker action routine sets 9C to nonzero once the first dipole of a word marker has been received at the end of a byte. If 9C is nonzero then the second dipole in the word marker should find 9C is nonzero and thus JMP FA60/FAAD to process this byte just read.

3. LDX A3, the count of the number of bits remaining to be read. If -1 (\$FF) then the parity bit for this byte has already been received.

BMI F988/F9E2 which JMPs to FA10/FA60, the word marker action routine. Branch if the parity bit has been received and thus set high bit on in A3. The branch to F988/F9E2 should occur after the word marker dipole in the word marker bit.

4. See if dipole time is 0:

LDX \$00. Default dipole value of 0.

ADC \$30. Add limit for 0 dipole. Before the ADC the accumulator still contains the value from step 1.

ADC B0. The time adjustment value.

CMP B1. Compare value in accumulator to current dipole time.

If the dipole is in the 0 range then BCS F993/F9ED if accumulator is greater than current dipole time to increment A9, the counter of 0/1. X register is 0.

Again, following in Figure 10-22 are some values of B0 and corresponding value in accumulator (converted to a time value by shifting left two bits and dividing by 1,022,370) that represents the 0/1 adjustable baseline.

**Figure 10-22. 0/1 Adjustable Baseline Times**

	B0: 252	253	254	255	0	1	2
0/1 adjustable baseline time: (in microseconds)	404	412	420	428	432	440	448

5. See if a dipole time is a 1:

INX. The X register now 1 for a 1 dipole.

ADC \$26. Add limit for 1 range.

ADC B0. Add time adjustment.

CMP B1. Compare to current dipole time.

BCS F997/F9F1. If the accumulator is greater than B1, a 1 dipole has been read. Branch to F997/F9F1 to decrement the 0/1 counter, A9.

Again, following in Figure 10-23 are some values of B0 and corresponding value in accumulator (converted to a time value by shifting left two bits and dividing by 1,022,370) that represents the 1/WM adjustable baseline.

**Figure 10-23. 1/Word Marker Adjustable Baseline Times**

	B0:	252	253	254	255	0	1	2
1/WM adjustable baseline time: (in microseconds)		544	556	568	580	584	596	608

6. Test for word marker dipole:

ADC \$2C. Add limit for word marker.

ADC B0. Add time adjustment.

CMP B1. Compare to current dipole time.

BCC F98B/F9E5. If current time for dipole is greater than the maximum time for a word marker, then this dipole is too long to be considered a 0, 1, or word marker. Branch to F98B/F9E5 to handle this error.

JMP FA10/FA60. If B1 is less than or equal to the value in the accumulator then have read a dipole that is considered a word marker. JMP FA10/FA60 to process this word marker.

Again, following in Figure 10-24 are some values of B0 and corresponding value in accumulator (converted to a time value by shifting left two bits and dividing by 1,022,370) that represents the WM/error adjustable baseline.

**Figure 10-24. Word Marker/Error Adjustable Baseline Times**

	B0:	252	253	254	255	0	1	2
WM/Error adjustable baseline time: (in microseconds)		708	724	740	756	760	776	792

## Set A8 If Bytes Are Being Received F98B/F9E5–F992/F9EC

### Called by:

BCC at F986/F9E0 in Determine if Dipole Time Represents Noise, 0, 1, or Word Marker, BNE at F9D0/FA20 if the final parity value in 9B is not zero, then there is a parity error, BNE at F9E9/FA39 if the two dipoles just read were 1, then an error has occurred—there should never be two consecutive 1 dipoles.

If bytes are currently being received, as indicated by a nonzero in B4, set A8 to this same nonzero value. In either case (bytes being received or not), branch to F9AC/FA06 which if B4 is nonzero checks to see if this IRQ interrupt was caused by a timer A/timer 1 timeout.

### Entry conditions:

B4 contains 0 if bytes are not currently being received (for example, if reading a leader), or a nonzero value if bytes are currently being received.

### Exit conditions:

If B4 is nonzero at entry, then A8 is set to this same nonzero value.

### Operation:

1. LDA B4.
2. If B4 contains 0, branch to F9AC/FA06.
3. If B4 is nonzero then STA A8, indicating either a parity error, mismatch of dipole error, or a too-long dipole error has occurred for this byte.
4. Branch to F9AC/FA06.

## Increment or Decrement the 0/1 Balanced Counter F993/F9ED–F998/F9F2

### Called by:

BCS at F975/F9CF if a 0 dipole was read; alternate entry at F997/F9F1 by BCS at F97E/F9D8 if a 1 dipole was read, JMP/BPL at FA1C/FA6A if a word marker dipole was just read, but A3 indicates it is not expecting a word marker yet, then treat the dipole as a 1.

Either increment or decrement the 0/1 balanced counter A9 depending on whether the dipole just read was a 0 dipole or a 1 dipole.

When reading the all leader dipoles (which are 0 dipoles) this counter is incremented to 16 the tape read routines recognize that a leader is being read.

When reading bytes, this counter should balance at zero after every is bit read, as each bit should have a 0 dipole and a 1 dipole if a 0 data bit or a 1 dipole and a 0 dipole if a 1 data bit.

### **Entry conditions:**

A9 contains relative count of number of 0 and 1 dipoles read.

### **Exit conditions:**

A9 incremented if a 0 dipole was read, or decremented if a 1 dipole was read.

### **Operation:**

1. If the normal entry point then a zero dipole was just read. Increment A9, then branch to F999/F9F3.
2. F997/F9F1: A one dipole was just read. (Or a word marker dipole within a byte). Decrement A9, then fall through to F999/F9F3.

## **Determine Value to Adjust Baseline Times**

### **F999/F9F3-F9A1/F9FB**

#### **Called by:**

Fall through from F997/F9F1 if a 1 dipole was just read, BCS at F995/F9EF if a 0 dipole was just read.

What this routine appears to be doing is determining for each dipole how much the dipole ranges from the 0/1 timebase that determines whether dipoles read are a 0 or a 1.

If the net variation between dipole values read and the 0/1 timebase line falls below the 0/1 timebase line, 92 is set to a positive value at the end of reading the two dipoles, and later used in the Adjust B0 routine to decrease the 0/1 timebase line and the other timebase lines for the next bit to be read.

If the net variation between dipole values read and the 0/1 timebase line falls above the 0/1 timebase line, 92 is set to a negative value (high bit on) at the end of reading the two dipoles, and later used in the Adjust B0 routine to increase the 0/1 timebase line and the other timebase lines for the next bit to be read.

## Tape I/O Routines

During reads of data bits when you get first a 0 or 1 dipole and then just the opposite dipole, the above description is true. However, when reading all 0 dipoles such as are found in the leader, the tape read is not actually comparing the net variation to the 0/1 timebase line.

When reading leader dipoles, the reading for each dipole is compared to the 0/1 timebase line — 76 microseconds. If the total variation for the two leader dipoles falls below this line, 92 is set positive to provide for a decrease in the timebase lines for the next dipole. If the total variation for the two leader dipoles falls above this line, 92 is set negative (high bit on) for an increase in the timebase lines for the next bit.

To arrive at this value of 76 microseconds variation from the 0/1 timebase line, just convert \$13, the value in the SBC \$13 instruction, to its equivalent time in microseconds. Thus  $\$13 = 0001\ 0011$ , shift left two bits =  $0100\ 1100 = \$004C = 76$  (decimal) =  $78/1022370 = 76.2$  microseconds.

When reading the data bits, this 76 microseconds is subtracted from the 0/1 timebase line when reading a 0 dipole, and from the 1/word marker timebase line when reading a 1 dipole. Normally, at the start of tape read, the 1/word marker timebase line is 584 microseconds, while the 0/1 timebase line is 432 microseconds.  $584 - 432 = 152$ , which is equal to  $76 \times 2$ . Thus, the actual baseline compared to when reading data bits is the 0/1 timebase line. See the description in the variable 92 for some typical values for the time base lines.

This routine is executed twice for each bit read. Before reading the first dipole 92 has been reset to 0. Thus the total time in 92 at the end of each bit is just the differences for the two dipoles for this bit.

Here are a few examples of how this routine works:

If reading leader dipoles, at the start of the tape read the 0/1 timebase line is 432 microseconds.

$$\begin{array}{r} 432 \text{ microseconds} \\ - \quad 76 \text{ microseconds} \\ \hline 356 \text{ microseconds} \\ - \quad 347 \text{ expected time for leader dipole} \\ \hline 9 \end{array}$$

The value 9 (decimal) stored in 92 at end of first dipole. The second dipole for the leader would similarly obtain a difference of 9, for a total difference of 18 (decimal) in 92 for the



bit, which would result in a decreased 0/1 timebase line for the next bit read. Specifically, the 0/1 timebase line for the next bit would be 428 microseconds. Thus, the timebase lines are adjusted until finally the total for both dipoles in 92 gives a negative values.

For example, if the 0/1 timebase line was 404 microseconds, then

$$\begin{array}{r}
 404 \text{ microseconds} \\
 - \quad 76 \text{ microseconds} \\
 \hline
 328 \text{ microseconds} \\
 - \quad 347 \text{ expected time for leader dipole} \\
 \hline
 - \quad 19
 \end{array}$$

Repeating this for the second dipole, the total at the end of reading the bit is  $-38$ , or \$DA, which has the high bit on and would thus cause a BMI.

## Entry conditions:

92 contains 0 before read of first dipole. After reading the first dipole, 92 contains the baseline time for dipole  $- 76 -$  time just read for dipole. The accumulator contains the current baseline time (0/1 baseline or 1/word marker baseline) for the dipole just read. B1 contains a value equivalent to the time for dipole just read.

## Exit conditions:

After reading both dipoles (a bit), 92 contains (baseline time for first dipole read  $- 76 -$  time for first dipole )  $+$  (baseline time for second dipole read  $- 76 -$  time for second dipole).

## Operation:

1. At entry the accumulator contains current baseline time (0/1 baseline or 1/word marker baseline) for the dipole just read.
2. SEC.
3. SBC \$13 (equivalent to 76 microseconds).
4. ADC 92. Will be zero when processing the first dipole. Will contain (baseline  $- 76$  microseconds  $-$  first dipole time) when processing the second dipole.
5. STA 92.
6. Fall through to F9A2/F9FC.

## **Flip Dipole Indicator Switch** **F9A2/F9FC-F9A9/FA03**

### **Called by:**

Falls through from F9A1/F9FB for each dipole read.

Flip the indicator for which dipole has been read. After the flip, the indicator A4 is equal to 1 if we just read the first dipole, while it's equal to 0 if we just read the second dipole.

### **Entry conditions:**

A4 contains 0 if the first dipole was just read, or 1 if the second dipole was just read.

### **Exit conditions:**

A4 contains 1 if the first dipole was just read, or 0 if the second dipole was just read.

### **Operation:**

1. LDA A4.
2. EOR \$01.
3. STA A4, thus now A4 has been flipped in value.
4. BEQ F905/FA25. Branch if the result in A4 is zero indicating the second dipole just read.
5. Fall through to F9AA/FA04 to store the value of the dipole just read as the value of this bit if the first dipole was read.

## **Store Dipole Value as Bit** **F9AA/FA04-F9AB/FA05**

### **Called by:**

Falls through from F9A9/FA03 after reading first dipole.

Save the value of the first dipole, which is also taken to be the value of the bit, in B7.

### **Entry conditions:**

The X register contains the value of the dipole just read, either 0 or 1.

### **Exit conditions:**

The dipole value is stored in B7.

### **Operation:**

1. STX D7. Fall through to F9AC/FA06.

### **Check Possible Error and See If Receiving Bytes F9AC/FA06-F9AF/FA09**

#### **Called by:**

Fall through from F9AA/FA04, Store Dipole Value as Bit, BCS at F960/F9BA if dipole just read is considered noise, BEQ at F98B/F9E7 if an error occurred (too long dipole, mismatched dipoles, or parity error) and not receiving bytes, BNE at F991/F9EB if an error occurred (too long dipole, mismatched dipoles, or parity error) and receiving bytes, BMI at F9ED/FA3D if two consecutive zero dipoles were just read and A9 is \$80 or more, setting limit on number of leader dipoles counted, BCC at F9F1/FA41 if leader dipoles are being read, but 16 of them have not yet been read, BCC at F9F5/FA45 if leader dipoles are being read, and 16 or more have now been read.

If B4 is zero, indicating that the tape read routines are between blocks or before the first block, branch to F9D2/FA22.

If B4 is nonzero, indicating that the tape read routines are ready to receive bytes, fall through to F9B0/FA0A to see if this interrupt was caused by timer A/timer 1.

#### **Operation:**

1. LDA B4.
2. BEQ F9D2/FA22.
3. Fall through to F9B0/FA0A if B4 is nonzero.

### **Determine If Interrupt Was Caused by Timer A/Timer 1 Timeout F9B0/FA0A-F9C8/FA18**

#### **Called by:**

Falls through from F98E/FA08 after an error has occurred while reading a byte.

An interrupt has occurred while reading a byte that is an error, either a too long dipole, mismatched dipoles, or parity error. Test if the interrupt was caused by a timer A/timer 1 timeout. If not, just restore registers and RTI. If the interrupt was caused by a timer A/timer 1 timeout, consider the read of this bit complete. Then process the bit either as a normal data bit by branching to F9F7/FA47 or as a word marker by branching to F988/F9E2 which JMPs to FA10/FA60.

Timer A/timer 1 is set to a value by the routine that set timer A/timer 1 to a value to lag behind the FLAG/CA1 interrupt. Timer A/timer 1 interrupts are enabled after the word marker following a group of leader dipoles is read. If the normal FLAG/CA1 interrupt does not occur when reading the tape before timer A/timer 1 counts down to zero, the timer A/timer 1 timeout will cause the interrupt. Thus, timer A/timer 1 is used to allow possible recovery when reading bytes if for some reasons a FLAG/CA1 interrupt does not occur as expected.

Also reset A4 to zero to indicate the next dipole read is the first dipole of a bit.

### **Exit conditions:**

A4 contains 0, indicating the next dipole read is the first dipole of a bit.

### **Operation:**

1. 64: See if 02A3, CIA #1 interrupt log, has bit 0 = 1, indicating a timer A interrupt occurred. If bit 0 = 1 then branch to step 2. If bit 0 = 0, indicating no timer A interrupt, also test 02A4 to see if either FLAG or timer A interrupts are enabled. If either FLAG or timer A interrupts are enabled then restore the register and RTI. If neither FLAG nor timer A interrupts are enabled, fall through to step 2.

VIC: BIT 912D to test bit 6 of VIA #2 interrupt flag register. Bit 6 is the timer 1 flag.

If no timer 1 interrupt occurred then branch to FA22 which jumps to FF56 to restore registers and RTI.

2. Set A4 to 0.
3. 64: Set 02A4 to 0.
4. LDA A3.
5. Branch if A3 greater than or equal to 0 indicating data bits still being read.
6. If A3 = -1, parity bit has been received. Treat value just read as a word marker.

## **Determine If Parity for Byte Read Is Correct F9C9/FA19-F9D4/FA24**

### **Called by:**

BMI at FA02/FA52 if decrementing A3 leaves it = -1, indicating parity bit just read; alternate entry at F9D2/FA22

## Tape I/O Routines

branched to by several instructions which use it to JMP to FEBC/FF56 to restore registers and RTI. These instructions are located at F9AE/FA08, F9FE/FA4E, and F9BA/FA0D.

Determine if the parity for the byte just read is correct. If the final parity value in 9B, the parity work byte, is zero then parity is correct, else indicate an error. Odd parity is used, where the total number of ones including the parity bit is odd. The second half of the dipole is actually used in computing the parity on the tape read.

Also, set timer A/timer 1 value for next dipole.

Follow this example to see how parity is calculated and has to be zero at end of byte read to be correct.

If data byte written on tape was the value 3, the first dipole bit pattern looks like this:

										parity bit
Data byte	0	0	0	0	0	1	1	1	0	
Second dipole	1	1	1	1	1	0	0	0	1	
9B	0	0	0	0	0	0	0	0	0	

(note: 9B initialized to 0 before reading each byte)

accumulator	1	1	1	1	1	0	0	0	(from the second dipole for the bit)	
EOR 9B	0	1	0	1	0	1	1	1		
STA 9B	1	0	1	0	1	1	1	1		

The parity bit (for second dipole) is 1, thus 1 EOR 9B (now 1), gives the final result in 9B of zero.

Remember when doing these parity calculations for tape read that the second dipole is the actual value used for the Exclusive OR with 9B.

A double bit error (where an even number of bits have been transposed in value) can occur during read of a byte. If such an error occurs, the parity calculation will not show any error, as the total number of 1's is still odd. Thus a byte can be loaded that is in error, and it will not be corrected in the second block read as no error was flagged for it during the first block. However, you are notified of the error in the checksum computation at the end of the load which does an Exclusive OR for the entire area loaded.

### Entry conditions:

9B contains 0 if the parity calculated for the byte just read is correct, or 1 if the parity calculated for the byte just read is in error.

### Exit conditions:

Timer A/timer 1 is reset and timer A/timer 1 interrupts are enabled.

### Operation:

1. LDX \$A6 , prepare value used in setting timer A/timer 1
2. JSR F8E2/F95D to set CIA #1 timer A/VIA #2 timer 1 value.
3. LDA 9B.
4. If 9B is nonzero then a parity error has occurred. Branch to F98B/F9E5 to handle this error.
5. If 9B is zero, this byte was read without any parity errors.
6. F9D2/FA22: JMP FEBC/FF56 to restore registers and RTI.

### Set Adjustable Baseline Values for Next Bit F9D5/FA25-F9E3/FA33

#### Called by:

BEQ at F9A8/FA02 in Convert Time Between FLAG/CA1 Interrupts into One-Byte Value.

If 92 has its high bit on, indicating a need for more baseline time, increment B0.

Conversely, if 92 does not have its high bit on, indicating a need for less baseline time, decrement B0 .

If 92 is 0, don't adjust the baseline time as it was a perfect match for the last bit read.

Following are two charts in Figures 10-25 and 10-26 that show how from an initial value of B0, the current value in the accumulator representing the adjustable baseline time for this dipole, and the calculated value in 92, that the tape read routines determine whether to increment, decrement or leave unchanged B0 for the next bit to be read. Figure 10-25 illustrates how this works when reading leader dipoles (all 0) while Figure 10-2 illustrates this when reading normal data bits. (containing a 1 dipole and a 0 dipole or a 0 dipole and a 1 dipole).

These values are computed using the values for NTSC VICs which are available in the U.S. See F999/F9F3-F9A1/F9FB for the actual instructions that calculate 92 for each bit. The values in the accumulator were derived by the following instructions:

```

LDA $B0
CLC
ADC #$3C  Noise/0 Adjustable Baseline
ADC #$30
ADC $B0   0/1 Adjustable Baseline
ADC #$26
ADC $B0   1/Word Marker Adjustable Baseline
ADC #$2C
ADC $B0   Word Marker/Error Adjustable Baseline
    
```

The values in the accumulator can be converted to an equivalent time in microseconds by shifting the values left two bits, adding 2, and dividing by 1,022,370. The values in B1 were actual values observed during tape load operations.

As an example of how to convert B1 to a time value: B1 for \$54 can be converted to a time value by: 0101 0100 00 = 01 0101 0000 = \$0150 = 336 decimal + 2 = 338/1,022,370 = 330.6 microseconds + 19.4 microseconds = 350.0 microseconds (vs. 349.8 predicted from tape write for leader dipole.)

The calculations in the chart are in hex, and for final B0, I show whether B0 is incremented or decremented.

**Figure 10-25. Reading Leader Bits**

Initial B0	Check for 0 Dipole or 1 Dipole	Accumulator	B1	92	Final B0
1	0	\$6E	\$54	\$10	DEC
0	0	\$6C	\$54	\$0C	DEC
-1 (\$FF)	0	\$6B	\$54	\$0A	DEC
-2 (\$FE)	0	\$69	\$54	\$06	DEC
-3 (\$FD)	0	\$67	\$54	\$02	DEC
-4 (\$FC)	0	\$65	\$54	\$FC	INC

Thus, when reading leader dipoles, B0 should stabilize between \$FD (253 decimal) and \$FC (252 decimal). When I stopped tape loads by hitting the keyboard STOP key and examined B0 during the 10 second leader that precedes the header, I found that B0 had actually stabilized at 252-253, just as predicted by the calculations above.

For a VIC using the PAL European standard clock, B0 may stabilize to a different value, but it definitely should stabilize to one or two values. These values should also be close to those on the 64 NTSC.

B0 can only be incremented or decremented to certain limits before the arithmetic involved in calculating 92 causes the adjustable baseline times to no longer be valid.

For example, the limit on incrementing B0 seems to be \$10 (16 decimal). If an initial value of \$11 is used to calculate the adjustable baseline times, you arrive at baseline times of \$4D \$8E \$25 \$02 for the noise/0, 0/1, 1/word marker, and word marker/error baselines respectively. Since the 1/word marker baseline is less than the 0/1 baseline these baselines are obviously invalid. Correspondingly, if you decrement B0 to \$D0 you get invalid baseline times of \$0C, \$0D, \$04, \$01 for the same respective baselines as mentioned above.

Figure 10-26 shows calculations when reading data bits.

The values in B1 were actual values observed in tape load operations. \$54 is equivalent to 350 microseconds for a 0 dipole (vs. 345.8 predicted during tape write). For the 1 dipole the equivalent time is 484.0 microseconds (vs. 502.4 predicted during tape write). This 484.0 microseconds is derived by \$76 = 0111 0110 after shifting left two bits = 01 1101 1000 = \$1D8 =  $472 + 2 = 474 / 1,022,370 = 463.6 \text{ microseconds} + 20.4 \text{ microseconds} = 484.0 \text{ microseconds}$ .

**Figure 10-26. Reading Data Bits**

Initial B0	Check for 0 Dipole or 1 Dipole	Accumulator	B1	92	Final B0
1	0	\$6E	\$54	\$15	DEC
	1	\$95	\$76		
0	0	\$6C	\$54	\$10	DEC
	1	\$92	\$76		
-1 (\$FF)	0	\$6B	\$54	\$0E	DEC
	1	\$91	\$76		
-2 (\$FE)	0	\$69	\$54	\$09	DEC
	1	\$8E	\$76		
-3 (\$FD)	0	\$67	\$54	\$04	DEC
	1	\$8B	\$76		
-4 (\$FC)	0	\$65	\$54	\$FD	INC
	1	\$88	\$76		

Thus, again B0 stabilizes at 252-253 when reading data bits.



### Entry conditions:

92 contains 0 if the last bit read exactly matched the predicted time of the bit (both dipoles) from the adjustable baselines, adjustable baselines do not need to be adjusted. 92 has its high bit on if, during last bit read, the values read for the two dipoles were greater than the current 0/1 adjustable baseline. (Note: If two leader dipoles were greater than 0/1 adjustable baseline  $\times 2 - 152$  microseconds.) 92 has its high bit off if during last bit read, the values read for the two dipoles was less than the current 0/1 adjustable baseline. (Note: If two leader dipoles were less than 0/1 adjustable baseline  $\times 2 - 152$  microseconds.) B0 contains 0 if the tape load is just starting, or a factor used in computing the adjustable baseline times for the last bit read if the load is already in progress.

### Exit conditions:

92 contains 0. B0 is incremented if more baseline time is needed for next bit, is decremented if less baseline time needed for next bit, or remains same if the previous baseline times matched the times for the bit just read.

### Operation:

1. LDA 92. See F999/F9F3-F9A1/F9FB for how this value is calculated.
2. If 92 contains 0 branch to step 5.
3. If 92 has its high bit off, decrement B0, and branch to step 5.
4. If 92 has its high bit on, increment B0, and fall through to step 5.
5. Reset 92 to 0 in preparation for the next bit read.
6. Fall through to F9E4/FA34.

### Determine If Two Dipoles Are Data, Error, or Leader Bit

F9E4/FA34-F9F6/FA46

### Called by:

Fall through from F9E3/FA33 in Set Adjustable Baseline Values for Next Bit.

See if the two dipoles just read represent different values (First dipole a 0, the second a 1, or vice versa). If the dipoles are different, a valid data bit has been received, so branch to F9F7/FA47 to process it. If the two dipoles are the same, the

values for the two dipoles must be 0 or else the bit is considered to be in error, since two 1 dipoles are never expected.

If the two dipoles are 0's, we have just read what is considered a leader bit. Once 16 of these leader bits are received, set 96 to the same value that is in A9 to indicate we are either before or between blocks.

96 has to be reset to a nonzero value between blocks to allow at FA46/FA91 the resetting of B5 to nonzero, which indicates that the tape load routine is before a block waiting for the word marker at the end of the leader bits.

### Entry conditions:

The X register contains the value of the second dipole read for this bit. D7 contains the value of the first dipole read for this bit. A9 contains the relative count of the number of 0 and 1 dipoles read. If leader (0) dipoles are being read, A9 will be incremented for each 0 dipole read.

### Exit conditions:

96 contains 0 if the leader is not yet recognized or if data is being read from the block. 96 contains 16-126 (decimal) once 16 leader bits read. Once 16 leader bits have been read, each successive leader bit causes 96 to be incremented up to a maximum value of 126.

### Operation:

1. CPX D7. At entry to this routine X register contains the value of the second dipole and D7 contains the value of the first dipole.
2. If above comparison is not equal, indicating for this bit the two dipoles have different values, branch to F9F7/FA47 as this bit is a valid data bit.
3. If above comparison is equal, the two dipoles must either be 1 or they must be 0.

If X register is nonzero (1), two 1 dipoles have been read. Two 1 dipoles are an error, as the tape write routines never purposely write two consecutive 1 dipoles for a bit. Branch to F98B/F9E5 to process this error. If leader is being read, the error is just ignored. However, if data from a block is being read, an error flag is set in A8.

4. If two 0 dipoles have been read, this bit just read is considered a leader bit.

If A9 has its high bit on, which it would if 128 (decimal) leader bits have been read, branch to F9AC/FA06 as

maximum value that can be stored in 96 is 127, although the maximum observed in 96 when PEEKing it during tape load was only 126

5. If less than 16 of these leader bits have been read, branch to F9AC/FA06.
6. If 16 or more of these leader bits have been received, store the count of leader bits, the 0/1 balanced counter A9 in 96, thus indicating the tape read routines recognize that a leader is being read. Then branch to F9AC/FA06.

### **Process Data or Parity Bit F9F7/FA47-FA0F/FA5F**

#### **Called by:**

BPL at F9C5/FA15 if bit received as a result of timer 1 interrupt; BNE at F9E6/FA36 if valid bit read.

Process the data bit just received. First, Exclusive OR the second dipole value with the parity working byte, 9B. Decrement the bit counter, A3, and if now -1 then the all 8 data bits have been received so branch to F9C9/FA19 to determine if parity is ok for this byte. If 8 data bits have not yet been received, rotate the bit just received into the high bit of BF, the byte being built.

Finally, JSR F8E2/F95D to reset timer A/timer 1, exit by JMP FEBC/FF56 to restore registers and RTI.

A byte is physically stored in low-to-high sequence on tape: bit 0 is followed by bit 1 which is followed by bit 3, and so on up to bit 8, which is followed by the parity bit and a word marker.

#### **Entry conditions:**

B4 is nonzero 0 if reading real data from a block, 0 if currently between blocks or before first block. D7 contains 0 or 1, depending on the value of the first dipole of the bit just read. BF contains the byte being built for the tape read.

#### **Exit conditions:**

9B is Exclusive ORed with the value of the second dipole. BF contains the byte being built for the tape read. If a data bit was received, that data bit has been rotated into the high bit of BF.

### Operation:

1. TXA. X register contains value for second dipole  
EOR 9B  
STA 9B. Update parity work byte.
2. If B4 is 0, indicating between blocks or before first block, branch to F9D2/FA22 which JMPs to FEBC/FF56 to restore registers and RTI.
3. If B4 is nonzero, indicating reading data from a block, continue.
4. Decrement A3, the counter of the number of bits remaining to be read for this byte.
5. If A3 is now -1, the bit just read is the parity bit, in which case branch to F9C9/FA19 to determine if parity is ok for this byte.
6. If A3 is 0-7, LSR D7, shifting the value for first dipole read for this bit into the carry flag of the status register.
7. ROR BF, thus rotating this bit from the carry flag into the high bit of BF.
8. LDX \$DA and JSR F8E2/F95D to set timer A/timer 1 value.
9. JMP FEBC/FF56 to restore registers and RTI.

### Process Word Marker Dipole FA10/FA60-FA1E/FA6B

#### Called by:

JMP at F988/F9E2 if word marker dipole was read (most likely the word marker dipole at the end of leader bits), after first dipole read following reception of a parity bit, or if a timer A/timer 1 interrupt occurs after a parity bit has been received.

Once the first dipole of a word marker is read, this routine is executed. It is also called after the first dipole that is read following a parity bit.

If the word marker is one following a group of leader bits, branch to FA1F/FA6C to set 9C to 1 indicating a byte is complete (or a new byte is next), 96 to 0 indicating block recognized and now ready to read data, B4 to nonzero indicating the tape read routines are ready to receive bytes, and B5 to 0 indicating that the tape read routines are actually reading bytes of data from a block.

If the word marker is one following a byte of data, in FA53/FAA0 the byte just received in BF is stored in BD to allow passing of this byte to the byte action routine.

### Entry conditions:

96 contains 0 if no block recognized yet, or if actually reading data from the block. 96 is nonzero if at least 16 leader bits have been read and the tape is either in the leader before the first block (of header or program) or between blocks 1 and 2 (of header or program). B4 contains 0 if between blocks or before the first block. B4 is nonzero when the tape load routines are ready to receive data bytes. A3, the count of the bits remaining to be read for a data byte, is nonnegative if reading data bits, or  $-1$  if a parity bit has been read.

### Exit conditions:

During this routine no variables or counters are modified.

### Operation:

1. LDA 96.
2. If 96 is 0, indicating no block recognized yet or actually reading data from the block, branch to step 5.
3. If 96 is nonzero then at least 16 leader bits have been read and the tape is either in the leader before the first block (of header or program) or between blocks 1 and 2 (of header or program).  
LDA B4.
4. If B4 contains 0 (between blocks or before the first block), branch to FA1F/FA6C to set flags indicating at the end of a leader and ready to start reading data from a block.  
If B4 is nonzero (the tape load routines are ready to receive data bytes), just fall through to step 5.
5. LDA A3.
6. If A3 is nonnegative (reading data bits), jump/branch to F997/F9F1 and consider this word marker dipole to be a 1 dipole.
7. If A3 is  $-1$  (parity bit has been read), branch to FA1F/fall through to FA6C as this is a word marker dipole following a parity bit.

### Word Marker Action

FA1F/FA6C-FA43/FA90

### Called by:

BEQ at FA16/FA64 If 96 is nonzero (at least 16 leader bits have been read and the tape is either in the leader before the first block or between blocks 1 and 2) and  $B4 = 0$  (between

blocks or before the first block); BMI at FA1A/fall through from FA6A. If B4 is nonzero (the tape load routines are ready to receive data bytes) and A3 is  $-1$  (parity bit has been read), since this is a word marker dipole following a parity bit.

Take action now that a word marker dipole has been received. Set 9C to 1 to indicate a byte has been received. If bytes are not being received (if inside a leader), set flags to indicate the end of the leader has been reached and block data is now expected for the next byte read.

### Entry conditions:

9C contains 0 (waiting for next byte or receiving of current byte). B4 contains 0 if between blocks or before the first block, or a nonzero value when the tape load routines are ready to receive data bytes.

96 contains 0 if no block recognized yet, or if actually reading data from the block. 96 contains a nonzero value if at least 16 leader bits have been read and the tape is either in the leader before the first block (of header or program) or between blocks 1 and 2 (of header or program).

### Exit conditions:

9C contains 1 (byte has been completely received). If B4 contains 0 and 96 is nonzero, A8 is set to a nonzero value, 96 is set to 0, timer A/timer 1 is set and timer A/timer 1 interrupts are enabled, and B4 is set to a nonzero value. Also, when falling through to the following routine B5 will be set to 0.

### Operation:

1. JSR F8E2/F95D to set timer A/timer 1 value.
2. Increment 9C to indicate byte has been completely received.
3. LDA B4.
4. Branch if B4 is nonzero to FA44/FA91 (the tape load routines are ready to receive data bytes).
5. If B4 is zero then between blocks or before the first block, continue with step 6.
6. LDA 96.
7. If 96 is zero (no block recognized yet, or actually reading data from the block) BEQ FA5D/FAAA which JMPs to FEBC/FF56 to restore registers and RTI.
8. If 96 is nonzero (at least 16 leader bits have been read and the tape is either in the leader before the first block or between blocks 1 and 2 ), continue with step 9.

9. STA A8, setting A8 to a nonzero value.
10. Set 96 to 0.
11. Enable timer A/timer 1 interrupts for CIA #1/VIA #2.
12. Set B4 to a nonzero value, indicating ready to receive data bytes from a block.
13. Fall through to FA44/FA91.

### **Determine If Dipole Is in Block or Leader FA44/FA91-FA52/FA9F**

#### **Called by:**

BNE at FA31/FA7E if B4 is nonzero. This way of entry is the one used to reset B5 to nonzero (before block of data waiting for word marker at end of leader bits) and B4 to zero (between blocks). Between blocks one and two, once 16 leader bits have been read 96 is set to nonzero. B4 is still nonzero when 96 is reset to nonzero. Then this branch at FA31/FA7E takes us to this routine which finds 96 nonzero, and thus stores a nonzero value in B5 and zero in B4.

Also, execution falls through from FA42/FA8F when a word marker dipole has been received at the end of a group of leader bits or at the end of a data byte. In this way of entry, 96 is always zero, and thus this routine sets B5 to zero (actually reading bytes of data from the block) and then branches to FA53/FAA0 to prepare to pass the byte read to the byte action routines.

This routine performs:

If between blocks, now reset B4 and B5 to indicate block data is not expected, that a leader is being read.

If inside a block of data, branch to FA53/FAA0 to prepare to pass the byte read in to the byte action routines.

#### **Entry conditions:**

96 is nonzero if reading leader bits between blocks one and two, or zero if the word marker dipole at the end of leader bits or end of a data byte has been read.

#### **Exit conditions:**

If 96 is zero, B5 is set to zero also, indicating actually reading bytes of data from a block. If 96 is nonzero, B5 is also set to a nonzero value (reading leader bits and waiting for word marker dipole at end of leader bits), B4 is set to zero (between blocks one and two), and timer A/timer 1 interrupts are disabled for CIA #1/VIA #2.

**Operation:**

1. LDA 96.
2. STA B5.
3. If 96 is 0, branch to FA53/FAA0 to prepare for passing the byte just received to the byte action routines.
4. If 96 is nonzero, continue with step 5.
5. Set B4 to 0.
6. Disable timer 1 interrupts for CIA #1/VIA #2.
7. Fall through to FA53/FAA0.

**Store Byte Received and Check Error Flags  
FA53/FAA0-FA5F/FAAC****Called by:**

BEQ at FA48/FA95 if 96 is 0 (word marker dipole at end of byte just read), Falls through from FA50/FA9D if 96 is nonzero, indicating reading leader bits.

Store the byte just read, contained in BF, into BD. The byte action routine can thus handle this byte when the second dipole for the word marker bit is read and passes control to the byte action routine.

Set B6 to nonzero if any tape errors occurred during the read of this byte.

**Entry conditions:**

BF contains byte just read. A8 is nonzero if a parity error, long dipole error, or two 1 dipoles within a bit were encountered; or zero if none of these errors occurred. A9 should be zero if a balanced number of 0 and 1 dipoles were read for this byte; otherwise an error in dipole reading occurred.

**Exit conditions:**

BD contains the byte built in BF. B6 is zero if no errors occurred during read of this byte, or nonzero if any errors occurred.

**Operation:**

1. Store byte received from BF into BD.
2. Test for any errors during read of byte and set B6 to nonzero if errors.  
    LDA A8.  
    ORA A9.  
    STA B6.
3. JMP FEBC/FF56 to restore registers and RTI.



## **Determine Action to Take for this Byte**

**FA60/FAAD-FA85/FAD2 and**

**FA8D/FADA-FA90/FADD**

### **Called by:**

JMP at F966/F9C0 After reading the second dipole of a word marker at the end of a data byte.

First reset flags to prepare for the next byte to be received and set A7 to indicate which block we are currently reading, block 1 or block 2. Then determine the action to be taken for this byte.

If AA contains 0, we're waiting for the first block count-down character to arrive. Fall through to FA91/FADE.

If AA is from 1-15 (decimal), block countdown characters are being read. Branch to FAA9/FAF6.

If AA contains \$40, valid block countdown characters have arrived, and this byte received is to be treated as a valid data byte. Branch to FAC0/FB0D.

If AA contains \$80, the first block has been loaded, and we're waiting for the second block. Branch to FA86/FAD3.

AA is initialized to 0 before starting to read the first header block, and again before starting to read the first program block.

### **Entry conditions:**

AA indicates the action to be taken with this as just described.

### **Exit conditions:**

A3 contains 8. A4 contains 0. A8 contains 0. 9B contains 0. A9 contains 0. 9C contains 0. Timer A/timer 1 of CIA #1/VIA #2 is set to count value. If BE is nonzero, A7 is set to value from BE, thus  $A7 = 2$  if first block being read,  $A7 = 1$  if second block being read. 90, status byte, is set to indicate a long block error if the routine is still trying to read data bytes after all of load area has been loaded from first block.

### **Operation:**

1. JSR FB97/FBDB to reset counters and variables as follows:  
A3 = 8, A4 = 0, A8 = 0, 9B = 0, A9 = 0.
2. Set 9C to 0.
3. LDX \$DA and JSR F8E2/F95D to set value for timer A/timer 1 of CIA #1/VIA #2.
4. LDA BE.
5. If BE contains 0, branch to step 7.

## Tape I/O Routines

6. Set A7 from value in BE; 2 if this is first block, 1 if this is the second block.
7. See if AA's high bit is on. If not, branch to FA8D/FADA, step 11.
8. If AA's high bit is on, (AA = \$80), the first block load is complete. Continue with step 9.
9. If B5 is nonzero, the tape load routine is reading the tape from the leader between blocks one and two. Branch to FA86/FAD3 to reset AA to 0 to start looking for block countdown characters and then JMP FEBC/FF56 to restore registers and RTI.
10. If B5 is zero, actually reading data bytes. LDX BE, decrement X register. If result is nonzero branch to FA8A/FAD7 to restore registers and RTI.
11. However, if the BE minus 1 is zero (which it would be after first block has been loaded), the tape read routine is trying to read data bytes after the first block has already completed. In this case JSR FE1C/FE6A to set the status for a long block, branch to FA8A/FAD7 to restore registers and RTI.
12. FA8D/FADA: Branch here from step 7 if AA < \$80.  
If AA contains \$40 then BVS FAC0/FB0D as actually receiving a valid data byte.
13. If AA's low nybble is nonzero (from 1-15), branch to FAA9/FAF6 as the byte just received is a block countdown character.
14. If AA is 0, we are still looking for the first block countdown character, so fall through to FA91/FADE to look for the first block countdown character.

### Check for Valid Block Countdown Characters

#### FA91/FADE-FAA4/FAF1 and FABA/FB07-FABF/FB0C

##### Called by:

Fall through from FA90/FADD if AA = 0.

Test the byte received to see if it's what we're expecting at this point, which is a block countdown character. For block one a countdown character should have its high bit on, while for block two the high bit should be off. If the block countdown character doesn't correspond to this block that is expected, reset AA to \$80 to indicate between blocks. If the block countdown character is ok, set AA to the value in its low nybble, which should be 9.

### Entry conditions:

B5 contains 0 if bytes are being read from the block. B5 contains a nonzero value while waiting for a word marker at end of leader bits. B6 is nonzero if an error occurred during read of this byte. AA contains 0 to indicate that the routine is looking for block countdown characters. A7 contains 2 if the first block is being read or 1 if the second block is being read. BD is the byte just read.

### Exit conditions:

AA contains \$80 if the byte read doesn't match the block countdown character pattern expected for this block. The byte just read is stored in AA if this byte was a valid block countdown character.

### Operation:

1. LDA B5. If B5 is nonzero, indicating tape read routine is waiting for the word marker at the end of leader bits, just branch to FA8A/FAD7 to restore registers and RTI. B5 is reset to nonzero only between blocks 1 and 2 (of header or program).
2. However, if B5 is zero, a word marker has been received at the end of leader bits, and this byte received is actual data from the block (most likely the first block countdown character). Continue with step 3.
3. Test B6 to see if the byte just read was in error. If so, branch to FA8A/FAD7 to restore registers and RTI.
4. If no errors for byte just read, through a somewhat convoluted section of code, see if the character just read can be considered a block countdown character for the first block with its high bit on, or for the second block with its high bit off.
5. The way this check for block countdown character corresponding to block expected at this point is done is as follows:

A7 is loaded into the accumulator and then shifted right into the carry. Now, A7 contains 2 if this is the first block (0000 0010) or 1 if this is the second block (0000 0001). Thus, the shift will leave the carry clear if this is the first block, or the carry set if this is the second block.

Load the byte just received, BD, into the accumulator and BMI to see whether its high bit is set. If the high bit is

1, the carry should also be clear (first block). If the high bit is 0, the carry should also be set (second block).

If the carry status does not agree with the high bit status of BD, this block countdown character is considered to be in error, in which case branch to FABA/FB07 to reset AA to \$80 to indicate still between blocks, branch to FA8A/FAD7 to restore registers and RTI.

If the carry status does agree with the high bit, the first valid block countdown character for this block has been read. Fall through to FAA9/FAF6.

### **Last Block Countdown Character FAA5/FAF2-FAB9/FB0C**

#### **Called by:**

BNE at FA8F/FADC if AA is not \$80, \$40, or 0, as tape read routines are reading block countdown characters, fall through from FAA7/FAF4 after an initial valid block countdown character has been received.

For each block countdown character, see if it is the last one (it is if it's 1). when the final block countdown character arrives, set AA to \$40 to indicate the next byte read will be an actual data byte to be loaded into the load area. Also JSR FB8E/FBD2 to reset the pointer to the load area (AC) to the start of the load area (C1).

#### **Entry conditions:**

AA contains 0 if the initial block countdown character has not yet been recognized. The low nybble of AA will contain 1-15 after the initial block countdown character is recognized (should actually be equal to 8-7-6-5-4-3-2 and finally 1, at which point all the block countdown characters have been received). (C1) points to the start of the load area.

#### **Exit conditions:**

AA is decremented. If AA contains 0 after decrement, reset AA to \$40 to indicate ready to receive data. If AA has been reset to \$40, set (AC) from (C1) and set AB to 0.

#### **Operation:**

1. Mask off the high nybble of the byte just received (to get rid of the high bit if it is set), and then store this byte in AA.
2. Decrement AA.

3. If AA is not yet zero, still receiving block countdown characters, so branch to FA8A/FAD7 to restore registers and RTI.
4. If AA is now decremented to zero, all of the block countdown characters have been read, and now the next byte read will be a valid data byte to be loaded into the load area.

Set AA to \$40 to indicate ready to receive actual data bytes.

5. JSR FB8E/FBD2 to reset the pointer (AC) from (C1).
6. Set AB to zero. This step is really unnecessary as AB is again zeroed before the checksum is computed for the load.
7. Branch to FA8A/FAD7 to restore registers and RTI.

### **Look for Initial Block Countdown Character FA86/FAD3-FA8C/FAD6**

#### **Called by:**

BNE at FA78/FAC5 If AA = \$80 (first block load is complete) and B5 is nonzero (tape read routine is reading leader bits between blocks 1 and 2).

The tape read routines are between the first and second blocks and have read at least 16 leader bits if this routine is called. Reset AA to 0 to again start looking for the initial block countdown character.

#### **Entry conditions:**

AA contains \$80 (load of first block is complete). B5 is nonzero (tape read routine has read at least 16 leader bits between blocks 1 and 2).

#### **Exit conditions:**

AA contains 0 (to look for the initial block countdown character for the second block).

#### **Operation:**

1. Set AA to 0.
2. Fall through to FA8A/FAD7 to JMP FEBC/FF56 to restore registers and RTI.

### **Common Exit Point for RTI FA8A/FAD7-FA8C/FAD9**

This instruction is used as a common exit point for routines in the byte action section. Branch here to then JMP FEBC/FF56 to restore register and RTI.

#### **Operation:**

1. JMP FEBC/FF56.

### **Valid Data Byte Received; Test for Short Block FAC0/FB0D-FACD/FB1A**

#### **Called by:**

BVS at FA8D/FADA if AA = \$40 indicating reading valid data bytes.

A valid data byte has been received. Test if a short block error has occurred, where the block has ended and leader bits are being read, and if so then set the short block status error.

#### **Entry conditions:**

AA contains \$40 (receiving valid data bytes). B5 contains 0 if actually reading data bytes from a block, or a nonzero value if the tape is between blocks waiting for the word marker at the end of the leader bits.

#### **Exit conditions:**

If B5 is nonzero, set 90 to indicate short block status; prepare for reset of AA to 0 to start looking for the initial block countdown character of the next block.

#### **Operation:**

1. LDA B5.
2. If B5 contains 0 (reading data from a block) branch to FACE/FB1B.
3. If B5 is nonzero then reading leader bits between blocks when actually the byte action routine is still expecting to be reading bytes from the block. JSR FE1C/FE6A to set short block status in \$90, prepare for setting AA to zero (look for initial block countdown character) and JMP FB4A/FB97 to do end of block processing.

## **Check for End of Load** **FACE/FB1B-FAD5/FB22**

### **Called by:**

BEQ at FAC2/FB0F if AA = \$40 (reading data bytes) and if B5 = 0 (reading bytes of data from a block).

See if the pointer to the location for the byte being loaded has reached the intended end of the load area. If so, jump to FB48/FB95 to do end of block processing. If not, branch to FAD6/FB23 to see which block is being loaded.

### **Entry conditions:**

(AC) points to the location for the byte currently being loaded.

(AE) points to the end of the load area + 1.

### **Exit conditions:**

Carry set if (AC) greater than or equal to (AE) (all bytes have been loaded).

Carry clear if (AC) less than (AE) (bytes remain to be loaded).

### **Operation:**

1. JSR FCD1/FD11 to compare (AC) to (AE).

If (AC) greater than or equal to (AE) then carry will be set on return.

If (AC) less than (AE), carry will be clear on return.

2. BCC FAD6/FB23. If not done loading all bytes for this block branch.
3. JMP FB48/FB95. Do end of block processing if final byte has been loaded.

## **Determine Block Being Read** **FAD6/FB23-FADA/FB27**

### **Called by:**

BCC at FAD1/FB1E in Check Whether All Bytes Have Been Loaded to (branch if still loading data bytes for this block).

See which block is being loaded. If block one, fall through to FADB/FB28. If block two, branch to FB08/FB55.

### **Entry conditions:**

A7 contains 2 if loading first block, or 1 if loading second block.

## Operation:

1. LDX A7.
2. DEX.
3. If result in X register is 0 ( $A7 - 1 = 0$ , which it would occur is  $A7 = 1$  indicating second block being loaded), branch to FB08/FB55.
4. If result in X register is nonzero, loading first block, in which case fall through to FADB/FB28.

## Load/Verify for Block One of Header or Program FADB/FB28–FB07/FB54

### Called by:

Fall through from FAD9/FA26 if A7 indicates block one is being loaded.

If 93 indicates we're doing a verify, just compare the byte read to the memory location pointed to by (AC). If not equal, set B6 to 1 to indicate a verify error.

If doing a load, first check B6 to see if any errors for the byte just read. If no errors for the byte, branch to FB3A/FBB7.

If an error is indicated for the byte just read, store the address of where the byte is to be loaded onto the stack and increment (by 2) the pass one error index, 9E.

### Entry conditions:

BD contains the byte just read. 93 contains 0 if this is a load, or 1 if this is a verify. For load, B6 contains 0 if no errors occurred for the byte just read.

### Exit conditions:

B6 contains 1 if an error was detected during a verify. If this is a load and B6 was nonzero at entry (indicating an error during the read of this byte), store (AC), address of byte being loaded, on the stack, indexed by 9E. 9E is then incremented by two.

### Operation:

1. Test 93 to see if doing a load or a verify. If doing load, branch to step 5. If doing verify, fall through to step 2.
2. LDA BD, the byte just read, and compare to byte in memory pointed to by (AC) .
3. If comparison is equal then this byte read verifies ok, branch to step 5.
4. If comparison is not equal, set B6 to 1 to flag this verify error, fall through to step 5.



5. LDA B6.
6. If B6 is zero indicating no error occurred during read (for tape load) or verify of this byte BD, branch to FB3A/FB87. At FB3A/FB87 load the byte if doing a load, and increment the pointer to the load area (for either load or verify).
7. If B6 is nonzero, indicating either a read error during tape load or a verify error, first see how many addresses of error locations have already been stored on the stack (storing upward from 0100).
8. If 31 error addresses have already been stored for 31 read (load) or verify errors, branch to FB3A/FB80 to set the status 90 to indicate an unrecoverable read error for load/verify.
9. If less than 31 errors have been encountered, store address of current byte being loaded/verified on the stack, with the low byte of the address stored on the stack sequentially before the high byte of the address. Memory locations 0100-013D are used to hold the read error addresses, with two bytes for the address of each byte considered in error.  

The instructions which store the error address on the stack are:

```
LDX 9E      ; index into stack for block 1 errors
LDA AD      ; high byte of address of current byte
STA 0101,X  ; save in stack
LDA AC      ; low byte of address of current byte
STA 0100,X  ; save in stack above high byte of address.
```
10. After storing the address of the byte in error on the stack then increment by two 9E so that it will index to next available location on stack in which to store the next error.
11. JMP FB3A/FB87 to go ahead and load this byte that is considered in error.

## Block Two Processing FB08/FB55-FB32/FB7F

### Called by:

BEQ at FAD9/FB26 if A7 indicates second block is being read.

See if all errors have already been corrected that were flagged in the first block. If so, just increment (AC) as all errors flagged in pass 1 have been corrected. If 31 errors have

been corrected, no more will be corrected as that is the limit of errors that can be flagged in reading block 1. This situation of reaching the maximum flagged in block 1 also results in just incrementing (AC).

If more errors remain to be corrected, see if the next byte to be loaded has the same address as the next address stored on the stack. If not, just increment (AC). However, if they do match, a byte has been read that matches one that was considered in error during the read of block 1. Attempt correction if a match occurs, increment by two the pointer to the stack for error correction 9F.

If trying to correct for verify, just see if the current byte loaded matches the one pointed to by (AC). If not, again set B6 to a verify error and then set status to unrecoverable read error.

If trying to correct for load, check B6 for a read error for the byte just read. If no errors for the byte, branch to FB3A/FB87 to load this byte. If any errors for this byte, set status to unrecoverable read error.

### **Entry conditions:**

BD contains the byte just read. 93 contains 0 if this is a load, or 1 if this is a verify. For a load, B6 contains 0 if no errors occurred while this byte was being read. (AC) points to the address of the byte being loaded/verified. 9E contains a count of the number of error addresses stored on stack during block 1 (actually,  $2 \times$  number of errors). The maximum number of errors allowed for block 1 is 31. 9F is the index into the stack during block 2 for the pointer to the next address to be corrected when reading block 2.

### **Exit conditions:**

If address pointed to on stack by 9F matches the address in (AC), the current memory location being loaded/verified, then:

If no errors occurred during the verify or read (load) of the byte, the byte in error pointed to by (AC) is replaced by the byte just read, BD.

If any errors occurred during the verify or read of the current byte, location 90 is set to indicate and unrecoverable read error.

9F incremented by two.

## Operation:

1. See if second block error index, 9F is equal to 9E.
2. If equal then branch to FB43/FB90 to increment the pointer to the load area (AC). 9E and 9F could be equal under three possible conditions: both are 0 (no errors during first block), 9E did not reach its limit during block 1 (all errors have been corrected), or 9E reached its limit of 31 errors and could not handle the next error (all errors corrected but not all errors were recorded during block 1).
3. If not equal, 9F is still less than 9E, indicating that more errors occurred during block 1 load that need to be corrected; continue with step 4.
4. See if next error address on the stack matches the address in (AC), and if not, branch to FB43/FB90 to increment (AC). Instructions that do this compare of addresses are:

X register already contains value of 9F.

**LDA AC** ; low byte of address of current byte (AC)

**CMP 0100,X** ; low byte of error address

**BNE FB90**

**LDA AD** ; high byte of address of current byte (AC)

**CMP 0101,X** ; high byte of error address

**BNE FB90**

5. If the error location address is the same as (AC) then the byte just read is one that was considered in error during block 1. Increment 9F by two as another error address from the stack has been processed.
6. If 93 indicates a verify, compare the byte just loaded to the byte in memory pointed to by (AC). If equal, branch to step 7.  
If not equal, set B6 to 1 which will later cause an unrecoverable read error to be recorded. Continue with step 7.
7. **LDA B6**. If a verify error was just flagged or if a read error occurred (for a load), fall through to FB33/FB80 to record an unrecoverable read error.
8. If B6 contains 0, indicating no errors in verify or read for this byte, branch to FB3A/FB87, which (if doing a load) loads this byte from the second block into the memory location pointed to by (AC) that was considered in error during the load for block 1.

## **Flag Unrecoverable Read Error** **FB33/FB80-FB39/FB86**

### **Called by:**

Fall through from FB32/FB7F during attempt at error correction for a byte flagged in error for block 1 if a read error also occurs for the same byte in block 2, BCC at FAF3/FB40 in Load/Verify for Block One of Header Or Program (if more than 31 errors detected during read of block 1).

If more than 31 errors occurred while loading block 1, 90 is set to \$10 to indicate an unrecoverable read error.

During block 2 processing, if (AC) matches the address on the stack of the next byte in error from pass 1, see if this byte was read without error for block 2. If an error occurred during the read of this byte for block 2, also set 90 to \$10 to indicate an unrecoverable read error.

### **Exit conditions:**

90 contains \$10 (actually \$10 is ORed with 90).

### **Operation:**

1. LDA \$10, then JSR FE1C/FE6A to set 90 by ORing \$10 with current value in 90.
2. Branch to FB43/FB90 to increment pointer to the load area.

## **Load Byte and Increment Pointer to Load Area** **FB3A/FB87-FB47/FB94**

### **Called by:**

BEQ at FAED/FB3A if the byte read during load for block 1 did not contain any errors, BEQ at FB31/FB7E if the memory location considered in error during block 1 is to be reloaded from byte read in block 2, JMP at FB05/FB87 if the byte read during load for block 1 did contain errors, store it anyway; alternate entry at FB43/FB90 by BEQ at FB0C/FB59 if 9E = 9F, BNE at FB13/FB60 if the next error location on stack doesn't match AC, BNE at FB1A/FB67 if the next error location on stack doesn't match AD, BEQ at FB2A/FB77 as the result of a successful verify during block 2 correction, BNE at FB38/FB85 in Flag Unrecoverable Read Error.

If doing a load then store the byte just read, BD, at location pointed to by (AC).

Then (for either load or verify) increment (AC).

### Entry conditions:

93 contains 0 if this is a load, or 1 if this is a verify. BD is the byte just read. (AC) points to the memory location to which this byte is to be loaded.

### Operation:

1. Test 93 to see if doing a verify or a load. If doing a verify, branch to step 4.
2. LDA BD, the byte just read.
3. STA in memory location pointed to by (AC).
4. FB43/FB90: JSR FCDB/FD1B to increment pointer to load area (AC).
5. Branch to FB8D/FBCF which jumps to FEBC/FF56 to restore registers and RTI.

### End-of-Block Processing

#### FB48/FB95-FB67/FBAB

#### Called by:

JMP at FAD3/FB20 if (AC) is greater than or equal to (AE); alternate FB4A/FB97: JMP at FACB/FB18 if short block status is signaled.

End of block has been determined by (AC) becoming equal to (AE). Set AA to \$80 to indicate between blocks.

See if A7 indicates we just finished reading block 2. If so, we're all done, so branch to FB68/FBAC.

If we just finished reading block one, see if 9E indicates any errors during load of block one. If no errors during block one, set BE to 0. Setting BE to 0 prevents any long block errors from being flagged, but the load for block 2 is not skipped.

One way to speed up tape loads would be to stop the tape load if block 1 is loaded without any errors. If block 1 is loaded without errors, block 2 is read anyway to make sure the tape positions correctly at the end of the second block upon completion of the load, reducing the likelihood of writing over the second block during a subsequent save.

Also, this read of block 2 is necessary when reading sequential files.

However, when reading a program tape, if block one completes with no read errors, you could branch to FB68/FBAC to compute the checksum for block one. If this computed checksum matches the checksum at the end of the first block

saved on tape, this load would be considered a complete success. This method would cut the time for tape load almost in half, although the tape would be left positioned after the first block.

## Entry conditions:

BE contains 2 if finished reading block one, 1 if finished reading block two, or 0 if finished reading block 2 after reading a block 1 that had no errors. A7 contains 2 if finished reading the first block, 1 if finished reading the second block.

## Exit conditions:

AA contains \$80. BE is decremented from 2 to 1 or from 1 to 0. A7 is decremented from 2 to 1 or from 1 to 0. BE is set to 0 if finished reading block one and no errors were detected.

## Operation:

1. LDA \$80.
2. FB4A/FB97: STA AA to set AA to \$80 between blocks or to \$00 if short block error.
3. 64: Disable interrupts for CIA #1 timer A.
4. See if just finished reading block 2 and possibly decrement BE. Following code shows the sequence for the VIC.

	Possible Values of BE
LDX BE	2 1 0
DEX	1 0 -1
BMI FBA0	
STX BE	1 0

	Possible values of A7
	2 1
FBA0 DEC A7	1 0

BEQ FBAC. If finished with read of block 2, branch to FBAC.

5. LDA 9E.
6. If nonzero, errors occurred during block 1 read. Branch to FB8D/FBCF to jump to FEBC/FF56 to restore registers and RTI.
7. If zero, reset BE to 0. It seems the function of this setting BE to 0 is to prevent long block errors.
8. Branch to FB8B/FBCF to jump to FEBC/FF56 to restore registers and RTI.

## **Tape Load Completed** **FB68/FBAC-FB8D/FBD1**

### **Called by:**

BEQ at FB5E/FBA2 in End-of-Block Processing, if A7 decrements to 0, indicating second block read complete.

Both blocks have been processed. Reset the IRQ vector to its normal default setting. Then compute a checksum (or parity) over all the bytes that were just loaded. This checksum should be equal to the checksum that was just read as the last byte of block 2. If not equal, set status to indicate checksum error.

### **Entry conditions:**

BD contains the checksum read from the second block.

### **Operation:**

1. JSR FC93/FCCF to:

disable IRQ interrupts,

turn off tape motor,

disable all CIA #1/VIA #2 interrupts,

VIC reset keyboard column scan to column 3,

VIC Set 912B for free running time,

Enable CIA #1 timer A/VIA #2 timer 1 interrupts

Set timer A/timer 1.

64: Make screen visible again.

Restore saved IRQ vector (029F) to active IRQ vector

(0314)

2. Compute checksum.

Restore (AC) to start of load area (C1) by JSR  
FB8E/FBD2.

Clear checksum workbyte, AB. Load each byte from the load area, then EOR AB, STA AB for each byte loaded.

JSR FCDB/FD1B to increment pointer to load area and  
JSR FCD1/FD11 to see when (AC) = (AE).

The final checksum AB computed over the load area should be the same as the one loaded from block 2 of the tape, now held in BD.

3. Compare checksums:

LDA AB. Checksum over area loaded

EOR 9B. Checksum from block 2

The result of this Exclusive OR is nonzero if the two checksums are different, or zero if they are the same.

If checksums are different then JSR FE1C/FE6A to OR the status 90 with \$20, indicating a checksum error.  
 4. JMP FEBC/FF56 to restore registers and RTI.

## **Set Timer A/Timer 1 Value to Lag Behind FLAG/CA1 Interrupt F8E2/F95D-F92B/F98D**

### **Called by:**

JSR at F9CB/FA1B in Determine If Parity for Byte Read Is Correct (first dipole), JSR at FA0A/FA5A in Process Data or Parity Bit (after second dipole of each data bit), JSR at FA2A/FA77 in Word Marker Action (after reading a word marker dipole), JSR at FA67/FAB4 in Determine Action to Take for this Byte (after reading the second dipole of a word marker bit—byte complete).

Set timer A/timer 1 of CIA #1/VIA #2 to a value to limit the amount of time the tape can be read before an IRQ interrupt occurs. The apparent purpose of setting this timer A/timer 1 value, which lags behind the normal FLAG/CA1 interrupt, is to make certain an IRQ interrupt occurs within a set period of time just in case of errors during tape read such as dropouts.

Following is an example of the calculations (for the VIC) for what value would be stored in timer 1 if B0 was 252 at entry. (B0 was 252–253 when examined during actual tape loads.) This example shows the calculation of a value for timer 1 following reception of the second dipole of a word maker when a byte is considered complete.

```
LDA B0    1111 1100 (252, $FC)
ASL
ASL
          1111 0000
ADC $B0   1111 1100
          1110 1100 (and carry set)
CLC
```



## Tape I/O Routines

For this example use the JSR from FAB4 with X register = \$DA (which is stored in B1)

```
ADC B1    1101 1010
STA B1    1100 0110

LDA $00
BIT B0    1111 1100
BMI F972  (Branch would occur.)
F972 ASL B1 1011 0100 (C = 1)
ROL      (accumulator = 0000 0001)
ASL B1    0110 1000 (C = 1)
ROL      (accumulator = 0000 0011)
```

Thus, the accumulator now contains \$03 and B1 contains \$68.

```
TAX
F979 LDA 9128
CMP $15
BCC F979
```

The carry will be set when the routine falls through.

Now that the accumulator and B1 have been converted into an equivalent two-byte time, add this time to the current value in timer 2. Timer 2 contains \$FFFF — time since timer 2 was last set to \$FFFF. For this JSR from FAB4 about 168 cycles have been executed since the setting of timer 2 and this read of Timer 2. Thus subtract 168 (hex \$A8) from \$FFFF to arrive at the current value of timer 2, or \$FFFF — \$00A8 = \$FF57. Now add the value in B1 to the low timer value and the value that was stored in the X register above to the high timer. Thus, \$FFF7

```
+ 1 (for carry set)
$FFF8
+ $0368
$02A0 (or decimal 672)
```

And  $674/1,022,370 = 659$  microseconds.

Thus timer 1 is set to a value that will cause an interrupt in 659 microseconds from now. Since the 1 dipole of the word marker is expected to about 526 microseconds, this provides enough time for a 1 dipole to be recognized, but then prevents the tape read from continuing without an interrupt once 659 microseconds have passed.

### Entry conditions:

B0 is a factor used in computing what values to be set for the adjustable baseline times for the next bit read. The X register contains:

\$A6 on entry from JSR at F9CB/FA1B.

\$DA on entry from JSR at FA0A/FA5A.

\$DA on entry from JSR at FA67/FAB4 .

$((B1 - \$93) + B0) \times 2$  on entry from JSR at FA77.

Timer B (DD06-DD07)/timer 2 (9128-9129) contains \$FFFF minus time since timer B/timer 2 last reset to \$FFFF at the start of the interrupt service routine for this tape read.

### Exit conditions:

Timer A/timer 1 set to value based on value of X register, B0, and timer B/timer 2 at entry to routine.

### Operation:

1. STX B1. X register value at entry contains one factor used in calculating value for timer A/timer 1.
2. See the previous discussion for an example of what the instructions in this routine do in calculating a value for timer A/timer 1.

It appears that B1 is set to a value  $((B0 \times 4) + B0) + X$  register at entry).

But then B1 is shifted two bits left with the two high bits being shifted into the two low bits of the accumulator.

Bit 2 of the accumulator can also be set if B0 had its high bit on.

3. Now that accumulator and B1 contain a value representing a time, add these two values to the current values in timer B/timer 2. Timer B/timer 2 is set to a value equal to \$FFFF — time since it was last set to \$FFFF, which was at the start of this IRQ interrupt service routine. To determine how long this has been, you can actually add up the cycle times for the instructions that led to the point where the JSR to this routine was executed.
4. Store the result of adding accumulator — B1 to timer B/timer 2 in timer A/timer 1, thus setting timer 1 to a value that will cause a timeout in a certain period of time, preventing the tape read routine from going without an interrupt beyond this time limit.
5. 64: Store value from 02A2 into 02A4 if DC0D does not have the FLAG interrupt bit set then JMP FF43 to execute the IRQ interrupt handler and then return control here.

## **Appendix A**

---

# **Commodore 64 and VIC-20 I/O and Video Control Registers**



## CIA #1 (Commodore 64)

HEX ADDRESS CIA #1

DC00 I/O DATA PORT A

DC01 I/O DATA PORT B

\* - DATA DIRECTION  
FOR PORT A IS GIVEN  
ONLY FOR KEYBOARD

\*\* - DATA DIRECTION  
FOR PORT B IS GIVEN  
ONLY FOR KEYBOARD

DC02 DATA DIRECTION PORT A \*

DC03 DATA DIRECTION PORT B \*\*

DC04 TIMER A LOW COUNT/LATCH

DC05 TIMER A HIGH COUNT/LATCH

DC06 TIMER B LOW COUNT/LATCH

DC07 TIMER B HIGH COUNT/LATCH

DC08 TOD CLOCK - 1/10 SECONDS

DC09 TOD CLOCK - SECONDS

DC0A TOD CLOCK - MINUTES

DC0B TOD CLOCK - HOURS

DC0C SERIAL I/O BUFFER

DC0D INTERRUPT DATA(R) REGISTER

DC0E INTERRUPT MASK(W) REGISTER

DC0F CONTROL REGISTER A

DC0F CONTROL REGISTER B

BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
KBRD C7 KBRD C6 PADDLE SELECT	KBRD C5 JOY2 RT JOY2 LF PADDLE2 PADDLE1 FIRE	KBRD C4 JOY2 RT JOY2 LF PADDLE2 PADDLE1 FIRE	KBRD C3 JOY2 RT JOY2 LF PADDLE2 PADDLE1 FIRE	KBRD C2 JOY2 RT JOY2 LF PADDLE2 PADDLE1 FIRE	KBRD C1 JOY2 RT JOY2 LF PADDLE2 PADDLE1 FIRE	KBRD C0 JOY2 RT JOY2 LF PADDLE2 PADDLE1 FIRE	KBRD C0 JOY2 RT JOY2 LF PADDLE2 PADDLE1 FIRE
KBRD R7 KBRD R6 TIMER B PULSE/ TOGGLE	KBRD R5 JOY1 RT JOY1 LF PADDLE2 PADDLE1 FIRE	KBRD R4 JOY1 RT JOY1 LF PADDLE2 PADDLE1 FIRE	KBRD R3 JOY1 RT JOY1 LF PADDLE2 PADDLE1 FIRE	KBRD R2 JOY1 RT JOY1 LF PADDLE2 PADDLE1 FIRE	KBRD R1 JOY1 RT JOY1 LF PADDLE2 PADDLE1 FIRE	KBRD R0 JOY1 RT JOY1 LF PADDLE2 PADDLE1 FIRE	KBRD R0 JOY1 RT JOY1 LF PADDLE2 PADDLE1 FIRE
0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1
NOTE: WRITE TO LATCH; READ FROM COUNT 1/60 SECOND IRQ INTERRUPT FOR KEYBOARD READ AND JIFFY CLOCK UPDATE ; ALSO FOR TAPE READ; ALSO USED FOR BASIC'S RND							
NOTE: WRITE TO LATCH; READ FROM COUNT TAPE WRITE TIMING; ALSO USED IN SERIAL I/O TIMING							
USED BY BASIC'S RND							
NOT USED BY KERNAL							
NOT USED BY KERNAL							
IRQ FLAG	FLAG1	SERIAL	TOD CLK	TIMER B	TIMER A	TIMER A	TIMER A
MASTER O-4	FLAG1 MASK	SERIAL MASK	TOD CLK MASK	TIMER B MASK	TIMER A MASK	TIMER A MASK	TIMER A MASK
TOD CLK FREQ O-60HZ	SERIAL MODE O-1-OUT O-2-LOCK	TIMER A LOAD FROM LATCHES IF 1	TIMER A RUN MODE ON PB6 SHOT O-CONT.	TIMER A OUTPUT ON PB6 SHOT O-CONT.	TIMER A OUTPUT ON PB6 SHOT O-CONT.	TIMER A OUTPUT ON PB6 SHOT O-CONT.	TIMER A OUTPUT ON PB6 SHOT O-CONT.
SET O-1-ALARM O-2-00-02 PULSES O-3-01-POSITIVE CNT O-4-10-TIMER A O-5-UNDERFLOW O-6-11-TIMER A O-7-UNDERFLOWS WITH O-8-CNT POSITIVE	TIMER B COUNT MODE O-1-POSITIVE CNT O-2-TRANSITIONS O-3-UNDERFLOW O-4-11-TIMER A O-5-UNDERFLOWS WITH O-6-CNT POSITIVE	TIMER B LOAD FROM LATCHES IF 1	TIMER B RUN MODE ON PB7 SHOT O-CONT.	TIMER B OUTPUT ON PB7 SHOT O-CONT.	TIMER B OUTPUT ON PB7 SHOT O-CONT.	TIMER B OUTPUT ON PB7 SHOT O-CONT.	TIMER B OUTPUT ON PB7 SHOT O-CONT.

# CIA #2 (Commodore 64)

HEX ADDRESS CIA #2

DD00 I/O DATA PORT A

DD01 I/O DATA PORT B (RS-232)

DD02 DATA DIRECTION PORT A

DD03 DATA DIRECTION PORT B

DD04 TIMER A LOW COUNT/LATCH

DD05 TIMER A HIGH COUNT/LATCH

DD06 TIMER B LOW COUNT/LATCH

DD07 TIMER B HIGH COUNT/LATCH

DD08 TOD CLOCK - 1/10 SECONDS

DD09 TOD CLOCK - SECONDS

DD0A TOD CLOCK - MINUTES

DD0B TOD CLOCK - HOURS

DD0C SERIAL I/O BUFFER

DD0D INTERRUPT DATA(R) REGISTER

DD0E INTERRUPT MASK(W) REGISTER

DD0F CONTROL REGISTER A

DD0F CONTROL REGISTER B

BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
SERIAL DATA IN	SERIAL DATA OUT	SERIAL CLOCK OUT	SERIAL ATN OUT	RS-232 DATA OUT	VIDEO ADDRESS 15 OUT	VIDEO ADDRESS 14 OUT	VIDEO ADDRESS 13 OUT
DSR	CTS	PIN J	RCOLINE	RI	DTR	RTS	RCDDATA
1	1	0	0	0	0	0	0
1	1	1	1	1	1	1	1
RS-232 SEND NOTE: WRITE TO LATCH, READ FROM COUNT							
RS-232 RECEIVE NOTE: WRITE TO LATCH, READ FROM COUNT							
NOT USED BY KERNAL							
NOT USED BY KERNAL							
IRQ FLAG	FLAG1	SERIAL	TOD CLK	TIMER B	TIMER A	TIMER B	TIMER A
MASTER 0-4	FLAG1 MASK	SERIAL MASK	TOD CLK MASK	TIMER B MASK	TIMER A MASK	TIMER B MASK	TIMER A MASK
FREQ PORT 1-50KHZ 0-60KHZ	TIMER A COUNTS 1-CNT 1-DUT 0-02	FORCE LOAD FROM LATCHES IF 1	TIMER A RUN MODE 1-DNE-PB6 1-TOGGLE 0-NO	TIMER A OUTPUT ON PB6 1-YES 1-TOGGLE 0-NO	TIMER A 1-START 1-STOP	TIMER B 1-START 1-STOP	TIMER B 1-START 1-STOP
SET 1-ALARM 0-TOD CLOCK	TIMER B MODE COUNT SELECT 00-02 PULSES 01-POSITIVE CNT TRANSITIONS 10-TIMER A UNDERFLOW TRANSITIONS 11-TIMER A UNDERFLOWS WITH CNT POSITIVE	FORCE LOAD FROM LATCHES IF 1	TIMER B RUN MODE 1-DNE-PB7 1-TOGGLE 0-NO	TIMER B OUTPUT ON PB7 1-YES 1-TOGGLE 0-NO	TIMER B 1-START 1-STOP	TIMER B 1-START 1-STOP	TIMER B 1-START 1-STOP

## VIA #1 (VIC-20)

HEX ADDRESS VIA #1

9110 I/O DATA PORT B (RS-232)

9111 I/O DATA PORT A

9112 DATA DIRECTION PORT B

9113 DATA DIRECTION PORT A

9114 TIMER 1 LOW

9115 TIMER 1 HIGH

RS-232

9116 TIMER 1 LOW SEND

9117 TIMER 1 HIGH

9118 TIMER 2 LOW RS-232

9119 TIMER 2 HIGH RECEIVE

911A SHIFT REGISTER

911B AUXILIARY CONTROL

911C PERIPHERAL CONTROL

911D INTERRUPT FLAG

911E INTERRUPT ENABLE

911F I/O DATA PORT A

(NO HANDSHAKING ON CA1, CA2)

CB1 - RS-232 RECEIVED DATA

CB2 - RS-232 TRANSMITTED DATA

CA1 - RESTORE KEY

BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0		
DSR	CTS	UNUSED	RCOLINE	R1	DTR	RTS	RCODATA		
SER ATN TAPE SW LP FIRE		JOY L	JOY DWN	JOY UP	SER DATA	SER CLK			
1	1	1	1	1	0	0	1		
0	1	1	1	1	1	1	1		
WRITE-LATCH READ-COUNTER AND RESET T1 INTERRUPT FLAG									
WRITE-LATCH/COUNTER, TRANSFER LOW LATCH TO LOW COUNTER AND									
RESET T1 INTERRUPT FLAG, READ-COUNTER									
WRITE-LATCH (DOES NOT AFFECT COUNT-DOWN IN PROGRESS): READ LATCH									
WRITE-LATCH (DOES NOT AFFECT COUNT-DOWN IN PROGRESS) AND									
RESET T1 INTERRUPT FLAG, READ LATCH									
WRITE-LATCH: READ-COUNT AND RESET T2 INTERRUPT FLAG									
WRITE-COUNTER, TRANSFER LOW LATCH TO LOW COUNTER, RESET T2									
INTERRUPT FLAG, READ-COUNT									
- NOT USED BY KERNEL									
TIMER 1 MODE		TIMER 2		SHIFT REGISTER MODE		PORT B PORT A			
		MODE				LATCH			
						CONTROL			
<--- CB2 MODE-----> CB1MODE <--- CA2 MODE-----> CA1MODE									
TRO	TIMER 1		TIMER 2		CB1		CB2		
STATUS	TIMEOUT		TIMEOUT		TRANS		COMPLETE TRANS		
MASTER		TIMER 1		TIMER 2		CB1		CB2	
CONTROL						SHIFT		CA1	
FOR 0-6								CA2	
SER ATN TAPE SW LP FIRE		JOY L	JOY DWN	JOY UP	SER DATA	SER CLK			

# Appendix A

## VIA #2 (VIC-20)

HEX ADDRESS VIA #2	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
9120 I/O DATA PORT B	KBRD C7	KBRD C6	KBRD C5	KBRD C4	KBRD C3	KBRD C2	KBRD C1	KBRD C0
	JOY0(1) TAPEWRT							
9121 I/O DATA PORT A	KBRD R7	KBRD R6	KBRD R5	KBRD R4	KBRD R3	KBRD R2	KBRD R1	KBRD R0
9122 DATA DIRECTION PORT B	0	0	0	0	0	0	0	0
9123 DATA DIRECTION PORT A	1	1	1	1	1	1	1	1
9124 TIMER 1 LOW 1/60 SEC. IRQ	WRITE-LATCH READ-COUNTER AND RESET T1 INTERRUPT FLAG							
9125 TIMER 1 HIGH INTERRUPT FOR KEYBOARD SCAN	WRITE-LATCH/COUNTER, TRANSFER LOW LATCH TO LOW COUNTER AND RESET T1 INTERRUPT FLAG, READ-COUNT							
9126 TIMER 1 LOW AND JIFFY CLK	WRITE-LATCH (DOES NOT AFFECT COUNT-DOWN IN PROGRESS); READ LATCH							
9127 TIMER 1 HIGH ; TAPE READ	WRITE-LATCH (DOES NOT AFFECT COUNT-DOWN IN PROGRESS) AND RESET T1 INTERRUPT FLAG; READ LATCH							
9128 TIMER 2 LOW TAPE WRITE:	WRITE-LATCH; READ-COUNT AND RESET T2 INTERRUPT FLAG							
9129 TIMER 2 HIGH SERIAL I/O	WRITE-COUNTER, TRANSFER LOW LATCH TO LOW COUNTER, RESET T2 INTERRUPT FLAG; READ-COUNT							
912A SHIFT REGISTER	- NOT USED BY KERNAL							
912B AUXILIARY CONTROL	TIMER 1 MODE	TIMER 2 MODE	SHIFT REGISTER MODE		PORT B LATCH CONTROL		PORT A LATCH CONTROL	
912C PERIPHERAL CONTROL	<--- CB2 MODE-----> CB1MODE <--- CA2 MODE-----> CA1MODE							
912D INTERRUPT FLAG	IRQ STATUS	TIMER 1 POP	TIMER 2 CB1 POP	CB2 TRANS	SHIFT TRANS	CA1 COMPLETE	CA2 TRANS	CA2
912E INTERRUPT ENABLE	MASTER CONTROL FOR 0-6	TIMER 1	TIMER 2 CB1	CB2	SHIFT	CA1	CA2	CA2
912F I/O DATA PORT A (NO HANDSHAKING ON CA1, CA2)	KBRD R7	KBRD R6	KBRD R5	KBRD R4	KBRD R3	KBRD R2	KBRD R1	KBRD R0

CB1 - SERIAL SRQ IN (NOT USED BY KERNAL)

CB2 - SERIAL SRQ OUT

CA1 - TAPE READ

CA2 - SERIAL CLOCK OUT

NOTE: KBRD CX - MEANS KEYBOARD COLUMN X

KBRD RX - MEANS KEYBOARD ROW X



# VIC- 20 Control Registers

AUXILIARY CONTROL REGISTER MODES		PERIPHERAL CONTROL REGISTER MODES	
TIMER 1 MODES:		3 2 1 BITS OF PCR FOR CA2 MODES	
ACR7	ACR6	7 6 5 BITS OF PCR FOR CB2 MODES	CA2 - CB2 MODES
0	0	0 0 SET CA2/CB2 INTERRUPT FLAG ON HIGH-TO-LOW TRANS OF CA2/CB2 INPUT.	
0	1	0 1 SET CA2/CB2 INTERRUPT FLAG ON R/W TO PORT A/B.	
1	0	0 0 1 SET CA2/CB2 INTERRUPT FLAG ON HIGH-TO-LOW TRANS OF CA2/CB2 INPUT.	
1	1	0 1 0 SET CA2/CB2 INTERRUPT FLAG BY R/W TO PORT A/B.	
TIMER 2 MODES:		0 1 0 SET CA2/CB2 INTERRUPT FLAG ON LOW-TO-HIGH TRANS OF CA2/CB2 INPUT.	
ACR5		0 1 1 SET CA2/CB2 INTERRUPT FLAG ON LOW-TO-HIGH TRANS OF CA2/CB2 INPUT.	
0	INTERVAL TIMER IN ONE-SHOT MODE	0 1 1 SET CA2/CB2 INTERRUPT FLAG BY R/W TO PORT A/B.	
1	COUNTS PULSES ON PB6	1 0 SET CA2/CB2 LOW ON WRITE TO PORT A/B. RESET CA2/CB2 HIGH WITH CA1/CB1 TRANSITION.	
SHIFT REGISTER MODES:		1 0 1 SET CA2/CB2 LOW FOR ONCE CYCLE AFTER A WRITE TO PORT A/B.	
ACR4	ACR3	1 1 0 HOLD CA2/CB2 OUTPUT LOW.	
0	0	1 1 1 HOLD CA2/CB2 OUTPUT HIGH.	
0	0	CA1 - CB1 MODES:	
0	1	4 BIT OF PCR FOR CB1 MODES	
0	1	0 BIT OF PCR FOR CA1 MODES	
1	0	0 CA1/CB1 INTERRUPT FLAG SET BY HIGH-TO-LOW TRANSITION OF CA1/CB1 INPUT SIGNAL AND CLEARED BY READ OR WRITE TO PORT	
1	1	1 CA1/CB1 INTERRUPT FLAG SET BY LOW-TO-HIGH TRANSITION OF CA1/CB1 INPUT SIGNAL AND CLEARED BY READ OR WRITE TO PORT	
PORT B LATCH CONTROL:			
ACR1			
0	INPUT LATCHING DISABLED - LATCHES REFLECT DATA ON THE PINS		
1	INPUT LATCHING ENABLED		
PORT A LATCH CONTROL:			
ACR0			
0	INPUT LATCHING DISABLED - LATCHES REFLECT DATA ON THE PINS		
1	INPUT LATCHING ENABLED		



## **Appendix B**

---

# **Index of Kernal Routines by Address**



# Index of Kernal Routines by Address

---

## Commodore 64

- E505-E509, Return Number of Columns and Rows in Screen, p. 179  
E50A-E517, Read/Plot Cursor Location, p. 179  
E518-E599, Set VIC Chip Registers, Clear Screen, Home Cursor, Set  
Screen Line Link Table, p. 129  
E59A-E59F, Set I/O Defaults and Home Cursor, p. 172  
E5A0-E5A7, Set Default Device Numbers, p. 133  
E5A8-E5B3, Initialize VIC Chip Registers, p. 134  
E5B4-E5C9, Retrieve Character from Keyboard Queue, p. 147  
E5CA-E631, Get Characters Until RETURN Key Detected, p. 142  
E632-E683, Get Character from Keyboard or Screen, p. 137  
E684-E690, If Quote Key Detected Then Flip Quote Flag, p. 145  
E691-E6A7, Display Screen Codes, p. 160  
E6A8-E6B5, Exit from Screen Editor Routines, p. 162  
E6B6-E700, Advance Cursor and Scroll or Insert Blank Lines, p. 162  
E701-E715, Move Cursor to Previous Screen Line, p. 165  
E716-E87B, Main Screen Editor, p. 148  
E87C-E890, Advance Cursor To Next Screen Line, p. 166  
E891-E8A0, Handle RETURN Key, p. 167  
E8A1-E8B2, Decrement Screen Line Pointer If Cursor Moves Left to New  
Line, p. 168  
E8B3-E8C1, Increment Screen Line Pointer If Cursor Moves Right to New  
Line, p. 169  
E8CB-E8D9, Test for Color Key, p. 170  
E8EA-E964, Scroll Screen, p. 171  
E965-E9C7, Insert Blank Line, p. 174  
E9C8-E9DF, Move Screen Line, p. 176  
E9E0-E9EF, Set Color Memory Pointers for Moving Line, p. 177  
E9F0-E9FE, Set Pointers to Screen Line Cursor Is On, p. 127  
E9FF-EA12, Clear Screen Line Cursor Is On, p. 126  
EA13-EA1B, Set Color and Store Character on Screen, p. 145  
EA1C-EA23, Display Byte in Accumulator on Screen, p. 145  
EA24-EA30, Set Pointer to Color Nybble, p. 146  
EA31-EA86, IRQ Interrupt Handler, p. 48  
EA87-EB47, Keyboard Scan, p. 53  
EB48-EB78, Keyboard Table Setup, p. 60  
EC44-EC4E, Test for Character Set Switch, p. 178  
ED09-ED10, Send TALK Command to Device, p. 214  
ED0C-ED10, Send LISTEN Command To Device, p. 193

## Appendix B

ED11-ED3F,	Do Attention Handshake With Serial Device, p. 194
ED40-EDAC,	Send Serial Byte: Command or Data, p. 197
EDAD-EDB8,	Set Status Word, p. 205
EDB9-EDBD,	Send Secondary Address After LISTEN, p. 204
EDBE-EDC6,	Bring Serial Bus Attention Line High, p. 190
EDC7-EDDC,	Send Secondary Address After TALK and Do TALK-LISTEN Turnaround, p. 214
EDDD-EDEE,	Send Serial Byte Deferred, p. 205
EDEF-EE12,	Send UNTALK Command, p. 222
EDFE-EE12,	Send UNLISTEN Command, p. 222
EE13-EE84,	Receive Byte from Serial Device, p. 216
EE85-EE8D,	Bring Serial Bus Clock Line High, p. 191
EE8E-EE96,	Bring Serial Bus Clock Line Low, p. 191
EE97-EE9F,	Bring Serial Bus Data Line High, p. 266
EEA0-EEA8,	Bring Serial Bus Data Line Low, p. 191
EEA9-EEB2,	Read Serial Data In and Serial Clock In, p. 192
EEB3-EEBA,	Delay One Millisecond, p. 206
EEBB-EED6,	Transmit RS-232 Bit: NMI Interrupt Driven, p. 260
EED7-EEF1,	Prepare Parity Bit and Set Counter for Stop Bits, p. 263
EF00-EF05,	Prepare to Send Stop Bit, p. 262
EF06-EF2D,	Prepare to Transmit Next Byte, p. 258
EF2E-EF49,	Handle Errors While Transmitting to RS-232 Device, p. 259
EF4A-EF58,	Compute Bit Count, p. 240
EF59-EF6D,	Receive RS-232 Bit: NMI Interrupt Driven, p. 245
EF6E-EF7D,	Check for Stop Bits, p. 250
EF7E-EF8F,	Prepare to Receive Next Byte, p. 253
EF90-EF96,	Check for Start Bit, p. 253
EF97-EFB2,	Store Byte Received into Buffer, p. 247
EFB3-EFC6,	Check Parity of Received Byte, p. 248
EFC7-EFDA,	Handle Errors While Receiving from RS-232 Device, p. 250
EFDB-EFE0,	Check for Framing/Break Error, p. 252
EFE1-F013,	Open RS-232 Channel for Output, p. 254
F014-F02D,	Store Character in Transmit Buffer, p. 256
F02E-F04C,	Prepare Timer A/Timer 1 Interrupt for RS-232 Transmission, p. 257
F04D-F085,	Open RS-232 Channel for Input, p. 241
F086-F0A3,	Get Character from RS-232 Receive Buffer, p. 244
F0A4-F0B6,	Disable RS-232 During Serial or Tape I/O, p. 266
F12B-F13D,	Display Kernal Messages, p. 113
F13E-F14D,	GETIN Preparation, p. 84
F14E-F156,	GETIN from RS-232 Device, p. 243
F157-F178,	Determine Input Device, p. 76
F157-F172,	CHRIN from Keyboard or Screen, p. 136
F179-F198,	CHRIN from Tape, p. 339
F199-F1AC,	Return Byte from Tape Buffer, p. 340
F1AD-F1B7,	Get Character from Serial Input Channel, p. 220
F1B8-F1C9,	CHRIN from RS-232 Device, p. 243
F1CA-F1DB,	Determine Output Device, p. 76
F1DC-F207,	CHROUT to Tape, p. 292
F208-F20D,	CHROUT to RS-232 Device, p. 255

## Appendix B

F20E-F236,	CHKIN Execution, p. 70
F237-F24F,	Open Serial Input Channel, p. 219
F250-F278,	CHKOUT Execution, p. 72
F279-F290,	Open Serial Output Channel, p. 221
F291-F2AA,	Determine Device for CLOSE, p. 80
F2AB-F2C7,	Close Logical File for RS-232 Device, p. 264
F2C8-F2ED,	Close Logical File for Tape, p. 295
F2EE-F2F0,	Close Logical File for Serial Device, p. 223
F2F1-F30E,	Common Exit for Close Logical File, p. 81
F30F-F31E,	See if Logical File Exists, p. 113
F31F-F32E,	Extract Logical File Number, Device Number, And Secondary Address from Tables, p. 115
F32F-F332,	Reset to No Open Files, p. 78
F333-F349,	Clear Serial Channels And Reset Default Devices, p. 82
F34A-F3D4,	OPEN Execution, p. 91
F38B-F398,	Determine If Open Is for Read or Write, p. 337
F399-F3D4,	Open Logical File for Reading from Tape, p. 337
F3B8-F3D4,	Open Logical File for Writing to Tape, p. 292
F3D5-F408,	Send OPEN, LOAD, or SAVE Command to Device, p. 202
F409-F482,	Open Logical File for RS-232 Device, p. 233
F483-F49D,	CIA Initialization for RS-232 (64), p. 239
F49E-F4A4,	Jump to LOAD Vector, p. 88
F4A5-F4B7,	Determine Device for LOAD, p. 88
F4B8-F532,	Load or Verify from Serial Device, p. 210
F533-F538,	Determine Device for LOAD, p. 88
F539-F5AE,	Control Routine for Tape Load, p. 341
F5AF-F5C0,	Display SEARCHING FOR Message, p. 118
F5C1-F5D1,	Display Filename, p. 118
F5D2-F5DC,	Display LOADING/VERIFYING Message, p. 115
F5DD-F5EC,	Jump to SAVE Vector, p. 100
F5ED-F5F9,	Determine Device for SAVE, p. 100
F5FA-F641,	Save to Serial Device, p. 207
F633-F639,	Stop Load or Save, p. 209
F642-F658,	Send Secondary Address for CLOSE, p. 210
F659-F68E,	Control Routine for Tape Save, p. 296
F68F-F69A,	Display SAVING Filename Message, p. 116
F69B-F6DC,	Jiffy Clock Update and STOP Key Scan, p. 51
F6DD-F6EC,	RDTIM/SETTIM Execution, p. 94
F6ED-F6FA,	Test for Stop Key, p. 107
F6FB-F72B,	Error Message Handler, p. 116
F72C-F769,	Find Next Tape Header, p. 347
F76A-F7CF,	Prepare Header and Write to Tape, p. 300
F7D0-F7D6,	Load and Check Tape Buffer Address, p. 299
F7D7-F7E9,	Set Start and End of Tape Buffer, p. 299
F7EA-F80C,	Find Specified Tape Header, p. 346
F80D-F816,	Increment Count of Number of Characters in Tape Buffer, p. 294
F817-F82D,	Display PRESS PLAY ON TAPE, p. 330
F82E-F837,	Check for Tape Button Down, p. 329
F838-F840,	Display PRESS RECORD & PLAY ON TAPE, p. 331

## Appendix B

F841-F849,	Read Tape Header into Buffer, p. 349
F84A-F863,	Load Next Two Blocks, p. 349
F864-F866,	Set Pointers to Start and End of Buffer and Write Buffer, p. 294
F867-F86A,	Prepare to Write Program to Tape, p. 302
F86B-F874,	Prepare IRQ Vector and Timer Interrupts for Tape Write, p. 302
F875-F8CF,	Reset IRQ Vector and Set Interrupt Enable Register, p. 303
F8D0-F8E1,	Check Keyboard STOP Key During Tape I/O, p. 331
F8E2-F92B,	Set Timer A/Timer 1 Value to Lag Behind Flag/CA1 Inter- rupt, p. 396
F92C-F939,	Determine Time Between Flag/CA1 Interrupts for this Dipole, p. 352
F93A-F95A,	Convert Time Between Interrupts into One-Byte Value, p. 355
F959-F98A,	Determine If Dipole Time Represents Noise, 0, 1, or Word Marker, p. 358
F98B-F992,	Set A8 If Bytes are Being Received, p. 362
F993-F998,	Increment or Decrement the 0/1 Balanced Counter, p. 362
F999-F9A1,	Determine Value to Adjust Baseline Times, p. 363
F9A2-F9A9,	Flip Dipole Indicator Switch, p. 366
F9AA-F9AB,	Store Dipole Value as Bit, p. 366
F9AC-F9AF,	Check Possible Error and See if Receiving Bytes, p. 367
F9B0-F9C8,	Determine if Interrupt Was Caused by Timer A/Timer 1 Timeout, p. 367
F9C9-F9D4,	Determine if Parity for Byte Read is Correct, p. 368
F9D5-F9E3,	Set Adjustable Baseline Values for Next Bit, p. 370
F9E4-F9F6,	Determine If Two Dipoles Are Data, Error, or Leader Bit, p. 373
F9F7-FA0F,	Process Data or Parity Bit, p. 375
FA10-FA1E,	Process Word Marker Dipole, p. 376
FA1F-FA43,	Word Marker Action, p. 377
FA44-FA52,	Determine if Dipole Is in Block or Leader, p. 379
FA53-FA5F,	Store Byte Received and Check Error Flags, p. 380
FA60-FA85,	Determine Action to Take for this Byte, p. 381
FA86-FA89,	Look for Initial Block Countdown Character, p. 385
FA8A-FA8C,	Common Exit Point For RTI, p. 386
FA8D-FA90,	Determine Action to Take for this Byte, p. 381
FA91-FAA4,	Check for Valid Block Countdown Characters, p. 382
FAA5-FAB9,	Last Block Countdown Character, p. 384
FAC0-FACD,	Valid Data Byte Received; Test for Short Block, p. 386
FACE-FAD5,	Check for End of Load, p. 387
FAD6-FADA,	Determine Block Being Read, p. 387
FADB-FB07,	Load/Verify for Block One of Header or Program, p. 388
FB08-FB32,	Block Two Processing, p. 389
FB33-FB39,	Flag Unrecoverable Read Error, p. 392
FB3A-FB47,	Load Byte and Increment Pointer to Load Area, p. 392
FB48-FB67,	End-of-Block Processing, p. 393
FB68-FB8D,	Tape Load Completed, p. 395
FB8E-FB96,	Reset Pointer to Start of Load/Save Area, p. 332
FB97-FBA5,	Reset Counters and Variables for Tape I/O, p. 333



## Appendix B

FBA6-FBC7,	Reverse Tape Write Line and Set Timer for Next Interrupt, p. 305
FBC8-FBCC,	Indicate Block Save Complete, p. 327
FBCD-FBEF,	Write a Word Marker Bit to Tape, p. 318
FBF0-FBF4,	Write Data Bit to Tape, p. 320
FBF5-FBFC,	Determine Which Part of Dipole Tape Write Routine Is Executing, p. 321
FBFD-FC0B,	Prepare to Write Second Dipole for This Bit, p. 322
FC0C-FC15,	Prepare to Write Next Bit and Decrement Bit Counter, p. 323
FC16-FC2F,	Prepare Counters for Next Byte and Test If Writing Block Countdown Characters, p. 324
FC30-FC3E,	Check for End of Tape Save, p. 325
FC3F-FC4D,	Move Next Byte from Save Area and Increment Pointer, p. 326
FC4E-FC56,	Prepare Parity Bit for This Byte, p. 326
FC57-FC69,	Handle End-of-Block Processing and Reset IRQ Vector, p. 328
FC6A-FC92,	Write Leader Bit to Tape and Reset IRQ Interrupt, p. 307
FC93-FCB7,	Reset CIA/VIA Registers and Restore IRQ Vector, p. 333
FCB8-FCC9,	Set IRQ Vector, p. 334
FCCA-FCD0,	Turn Off Tape Motor, p. 335
FCD1-FCDA,	Compare Pointer to Current Byte with Pointer for End of Load/Save, p. 335
FCDB-FCE1,	Increment Pointer to Current Byte, p. 336
FCE2-FD01,	System Reset, p. 17
FD02-FD0F,	Test for Autostart Cartridge, p. 18
FD15-FD25,	Initialize Kernal RAM Vectors, p. 22
FD50-FD9A,	Initialize Memory Pointers, p. 19
FDA3-FDF8,	Initialize I/O Chips, p. 24
FDF9-FDFF,	Set File Name Location And Number Of Characters, p. 105
FE00-FE06,	Set Logical File Number, Device Number, Secondary Address, p. 103
FE07-FE20,	Read/set Status-set Message Control, p. 97
FE21-FE24,	Set Serial Timeout Value, p. 223
FE25-FE33,	Memtop Execution, p. 90
FE34-FE42,	Membot Execution, p. 89
FE43-FE46,	NMI Interrupt Handler Jump, p. 33
FE47-FEC1,	NMI Interrupt Handler, p. 34
FE66-FE71,	BRK Interrupt Handler, p. 47
FED6-FF06,	NMI Interrupt Handler-Timer B Service, p. 37
FF07-FF2D,	NMI Interrupt Handler-Start Timer B for FLAG NMI, p. 37
FF48-FF5A,	IRQ/BRK Interrupt Switch, p. 45
FF5B-FF6D,	Initialize VIC-II Chip And Set PAL/NTSC Flag, p. 28
FF6E-FF80,	Initialize I/O Chips, p. 24
FF81, CINT,	p. 77
FF84, IOINIT,	p. 85
FF87, RAMTAS,	p. 93
FF8A, RESTOR,	p. 98
FF8D, VECTOR,	p. 109
FF90, SETMSG,	p. 104
FF93, SECOND,	p. 102
FF96, TKSA,	p. 108

## Appendix B

FF99, MEMTOP, p. 89  
FF9C, MEMBOT, p. 89  
FF9F, SCNKEY, p. 101  
FFA2, SETTMO, p. 106  
FFA5, ACPTR, p. 68  
FFA8, CIOUT, p. 78  
FFAB, UNTALK, p. 109  
FFAE, UNLSN, p. 108  
FFB1, LISTEN, p. 85  
FFB4, TALK, p. 107  
FFB7, READST, p. 95  
FFBA, SETLFS, p. 102  
FFBD, SETNAM, p. 104  
FFC0, OPEN, p. 90  
FFC3, CLOSE, p. 79  
FFC6, CHKIN, p. 69  
FFC9, CHKOUT, p. 71  
FFCC, CLRCHN, p. 82  
FFCF, CHRIN, p. 73  
FFD2, CHROUT, p. 75  
FFD5, LOAD, p. 86  
FFD8, SAVE, p. 98  
FFDB, SETTIM, p. 105  
FFDE, RDTIM, p. 94  
FFE1, STOP, p. 106  
FFE4, GETIN, p. 83  
FFE7, CLALL, p. 78  
FFEA, UDTIM, p. 108  
FFED, SCREEN, p. 101  
FFF0, PLOT, p. 93  
FFF3, IOBASE, p. 84

### **VIC-20**

E4A0-E4A8, Bring Serial Bus Data Line High, p. 190  
E4A9-E4B1, Bring Serial Bus Data Line Low, p. 191  
E4B2-E4BB, Read Serial Data In And Serial Clock In, p. 192  
E505-E509, Return Number Of Columns And Rows In Screen, p. 179  
E50A-E517, Read/Plot Cursor Location, p. 179  
E518-E5B4, Set Vic Chip Registers, Blank Screen, Set Cursor Pointers, Set Screen Line Link Table, p. 129  
E5B5-E5BA, Set I/O Defaults And Home Cursor, p. 113  
E5BB-E5C2, Set Default Device Numbers, p. 133  
E5C3-E5CE, Initialize Vic Chip Registers, p. 134  
E5CF-E5E4, Retrieve Keyboard Queue Character, p. 147  
E5E5-E64E, Get Characters Until Return Key Detected, p. 142  
E64F-E6B7, Get Character From Keyboard Or Screen, p. 137  
E6B8-E6C5, If Quote Key Detected Then Flip Quote Flag, p. 145  
E6C5-E6DB, Display Screen Codes, p. 160  
E6DC-E6E9, Exit From Screen Editor Routines, p. 162  
E6EA-E72C, Advance Cursor And Scroll Lines Or Insert Blank Lines If Needed, p. 162

## Appendix B

E72D-E741,	Move Cursor To Previous Physical Screen Line, p. 165
E742-E8C2,	Main Screen Editor, p. 148
E8C3-E8D7,	Advance Cursor To Next Screen Line, p. 166
E8D8-E8E7,	Handle RETURN Key, p. 167
E8E8-E8F9,	Decrement Screen Line Pointer If Cursor Moves Left to New Line, p. 168
E8FA-E911,	Increment Screen Line Pointer If Cursor Moves Right to New Line , p. 169
E912-E920,	Test for Color Key, p. 170
E975-E9ED,	Scroll Screen, p. 171
E9EE-EA55,	Insert Blank Line, p. 174
EA56-EA6D,	Move Screen Line, p. 176
EA6E-EA7D,	Set Color Memory Pointers for Moving Line, p. 177
EA7E-EA8C,	Set Pointers to Screen Line Cursor Is On, p. 127
EA8D-EAA0,	Clear Screen Line Cursor Is On, p. 126
AAA1-EAA9,	Set Color and Store Character On Screen, p. 145
EAAA-EAB1,	Display Byte In Accumulator On Screen, p. 145
EAB2-EABE,	Set Pointer to Color Nybble, p. 146
EABF-EB1D,	IRQ Interrupt Handler, p. 48
EB1E-EBDB,	Keyboard Scan, p. 53
EBDC-EC45,	Keyboard Table Setup, p. 60
ED21-ED2F,	Test For Character Set Switch, p. 178
EE14-EE1B,	Send TALK Command to Device, p. 214
EE17-EE1B,	Send LISTEN Command To Device, p. 193
EE1C-EE48,	Do Attention Handshake With Serial Device, p. 194
EE49-EEB3,	Send Serial Byte: Command or Data, p. 197
EEB4-EEBF,	Set Status Word, p. 205
EEC0-EEC4,	Send Secondary Address After LISTEN, p. 204
EEC5-EECD,	Bring Serial Bus Attention Line High, p. 190
EECE-EEE3,	Send Secondary Address After TALK and Do TALK-LISTEN Turnaround, p. 214
EEE4-EEF5,	Send Serial Byte Deferred, p. 205
EEF6-EF18,	Send UNTALK Command, p. 222
EF04-EF18,	Send UNLISTEN Command, p. 222
EF19-EF83,	Receive Byte from Serial Device, p. 216
EF84-EF8C,	Bring Serial Bus Clock Line High, p. 191
EF8D-EF95,	Bring Serial Bus Clock Line Low, p. 192
	And Serial Clock in.
EF96-EFA2,	Delay One Millisecond, p. 206
EFA3-EFBE,	Transmit RS-232 Bit: NMI Interrupt Driven, p. 260
EFBF-EFE7,	Prepare Parity Bit and Set Counter for Stop Bits, p. 263
EFE8-EFED,	Prepare to Send Stop Bit, p. 262
EFEE-F015,	Prepare to Transmit Next Byte, p. 258
F016-F026,	Handle Errors While Transmitting to RS-232 Device, p. 259
F027-F035,	Compute Bit Count, p. 240
F036-F04A,	Receive RS-232 Bit: NMI Interrupt Driven, p. 245
F04B-F05A,	Check for Stop Bits, p. 250
F05B-F067,	Prepare to Receive Next Byte, p. 253
F068-F06E,	Check for Start Bit, p. 253
F06F-F08A,	Store Byte Received into Buffer, p. 247

## Appendix B

F08B-F09E,	Check Parity of Received Byte, p. 248
F09F-F0B2,	Handle Errors While Receiving from RS-232 Device, p. 250
F0B3-F0B8,	Check for Framing/Break Error, p. 252
F0BC-F0EC,	Open RS-232 Channel for Output, p. 254
F0ED-F101,	Store Character in Transmit Buffer, p. 256
F102-F115,	Prepare Timer A/Timer 1 Interrupt for RS-232 Transmission, p. 257
F116-F14E,	Open RS-232 Channel For Input, p. 241
F14F-F15F,	Get Character From RS-232 Receive Buffer, p. 244
F160-F173,	Disable RS-232 During Serial or Tape I/O, p. 266
F1E2-F1F4,	Display KERNAL Messages, p. 113
F1F5-F204,	Getin Preparation, p. 84
F205-F20D,	GETIN from RS-232 Device, p. 243
F20E-F22F,	Determine Input Device, p. 75
F20E-F229,	CHRIN from Keyboard Or Screen, p. 136
F230-F24F,	CHRIN from Tape, p. 339
F250-F263,	Return Byte from Tape Buffer, p. 340
F264-F26E,	Get Character From Serial Input Channel, p. 220
F26F-F279,	CHRIN from RS-232 Device, p. 243
F27A-F28E,	Determine Device For CHROUT, p. 76
F28F-F2B8,	CHROUT to Tape, p. 292
F2B9-F2C6,	CHROUT to RS-232 Device, p. 255
F2C7-F2EF,	CHKIN Work, p. 118
F2F0-F308,	Open Serial Input Channel, p. 219
F309-F331,	CHKOUT Execution, p. 120
F332-F349,	Open Serial Output Channel, p. 221
F34A-F363,	Determine Device for CLOSE, p. 80
F364-F38C,	Close Logical File for RS-232 Device, p. 264
F38D-F3AD,	Close Logical File for Tape, p. 295
F3AE-F3B0,	Close Logical File For Serial Device, p. 223
F3B1-F3CE,	Common Exit For Close Logical File Routines, p. 81
F3CF-F3DE,	See If Logical File Exists, p. 114
F3DF-F3EE,	Extract Logical File Number, Device Number, And Secondary Address From Tables, p. 115
F3EF-F3F2,	Reset To No Open Files, p. 78
F3F3-F409,	Clear Serial Channels And Reset Default Devices, p. 82
F40A-F494,	OPEN Execution, p. 91
F44B-F458,	Determine If Open Is for Read Or Write, p. 337
F459-F494,	Open Logical File for Reading From Tape, p. 337
F478-F494,	Open Logical File for Writing to Tape, p. 292
F495-F4C6,	Send OPEN, LOAD, or SAVE Command to Device, p. 202
F4C7-F541,	Open Logical File for RS-232 Device, p. 233
F542-F548,	Jump To LOAD Vector, p. 88
F549-F55B,	Determine Device for LOAD, p. 88
F55C-F5C9,	Load or Verify from Serial Device, p. 210
F5CA-F5D0,	Determine Device for LOAD, p. 88
F5D1-F646,	Control Routine for Tape Load, p. 341
F647-F658,	Display SEARCHING FOR Message, p. 118
F659-F669,	Display Filename, p. 118
F66A-F674,	Display LOADING/VERIFYING Message, p. 115

## Appendix B

F675–F684,	Jump to SAVE Vector, p. 100
F685–F691,	Determine Device For SAVE, p. 100
F692–F6D9,	Save To Serial Device, p. 207
F6CB–F6D1,	Stop Load or Save, p. 209
F6DA–F6F0,	Send Secondary Address for CLOSE, p. 210
F6F1–F727,	Control Routine for Tape Save, p. 296
F728–F733,	Display SAVING Filename Message, p. 116
F734–F75F,	Jiffy Clock Update and STOP Key Scan, p. 51
F760–F76F,	RDTIM/SETTIM Execution, p. 94
F770–F77D,	Test For STOP Key, p. 107
F77E–F7AE,	Error Message Handler, p. 116
F7AF–F7E6,	Find Next Tape Header, p. 347
F7E7–F84C,	Prepare Header and Write to Tape, p. 300
F84D–F853,	Load and Check Tape Buffer Address, p. 299
F854–F866,	Set Start and End of Tape Buffer, p. 299
F867–F889,	Find Specified Tape Header, p. 346
F88A–F893,	Increment Count of Number of Characters in Tape Buffer, p. 294
F894–F8AA,	Display PRESS PLAY ON TAPE, p. 330
F8AB–F8B6,	Check for Tape Button Down, p. 329
F8B7–F8BF,	Display PRESS RECORD & PLAY ON TAPE, p. 331
F8C0–F8C8,	Read Tape Header into Buffer, p. 349
F8C9–F8E2,	Load Next Two Blocks, p. 349
F8E3–F8E5,	Set Pointers to Start and End of Buffer and Write Buffer, p. 294
F8E6–F8E9,	Prepare to Write Program to Tape, p. 302
F8EA–F8F3,	Prepare IRQ Vector and Timer Interrupts for Tape Write, p. 302
F8F4–F94A,	Reset IRQ Vector and Set Interrupt Enable Register, p. 303
F94B–F95C,	Check Keyboard STOP Key During Tape I/O, p. 331
F95D–F98D,	Set Timer A/Timer 1 Value to Lag Behind Flag/CA1 Inter- rupt, p. 396
F98E–F99B,	Determine Time Between Flag/CA1 Interrupts for this Dipole, p. 352
F99C–F9AF,	Convert Time Between Interrupts into One-Byte Value, p. 355
F9B0–F9E4,	Determine If Dipole Time Represents Noise, 0, 1, or Word Marker, p. 358
F9E5–F9EC,	Set A8 If Bytes are Being Received, p. 362
F9ED–F9F2,	Increment or Decrement the 0/1 Balanced Counter, p. 362
F9F3–F9FB,	Determine Value to Adjust Baseline Times, p. 363
F9FC–FA03,	Flip Dipole Indicator Switch, p. 366
FA04–FA05,	Store Dipole Value as Bit, p. 366
FA06–FA09,	Check Possible Error and See if Receiving Bytes, p. 367
FA0A–FA18,	Determine if Interrupt Was Caused by Timer A/Timer 1 Timeout, p. 367
FA19–FA24,	Determine if Parity for Byte Read is correct, p. 368
FA25–FA33,	Set Adjustable Baseline Values for Next Bit, p. 370
FA34–FA46,	Determine If Two Dipoles Are Data, Error, or Leader Bit, p. 373
FA47–FA5F,	Process Data or Parity Bit, p. 375

## Appendix B

FA60-FA6B,	Process Word Marker Dipole, p. 376
FA6C-FA90,	Word Marker Action, p. 377
FA91-FA9F,	Determine if Dipole Is in Block or Leader, p. 379
FAA0-FAAC,	Store Byte Received and Check Error Flags, p. 380
FAAD-FAD2,	Determine Action to Take for this Byte, p. 381
FAD3-FAD6,	Look for Initial Block Countdown Character, p. 385
FAD7-FAD9,	Common Exit Point for RTI, p. 386
FADA-FADD,	Determine Action to Take for this Byte, p. 381
FADE-FAF1,	Check for Valid Block Countdown Characters, p. 382
FAF2-FB0C,	Last Block Countdown Character, p. 384
FB0D-FB1A,	Valid Data Byte Received; Test for Short Block, p. 386
FB1B-FB22,	Check for End of Load, p. 387
FB23-FB27,	Determine Block Being Read, p. 387
FB28-FB54,	Load/Verify for Block One of Header or Program, p. 388
FB55-FB7F,	Block Two Processing, p. 389
FB80-FB86,	Flag Unrecoverable Read Error, p. 392
FB87-FB94,	Load Byte and Increment Pointer to Load Area, p. 392
FB95-FBAB,	End-of-Block Processing, p. 393
FBAC-FBD1,	Tape Load Completed, p. 395
FBD2-FBDA,	Reset Pointer to Start of Load/Save Area, p. 332
FBD8-FBE9,	Reset Counters and Variables for Tape I/O, p. 333
FBEA-FC05,	Reverse Tape Write Line and Set Timer for Next Interrupt, p. 305
FC06-FC0A,	Indicate Block Save Complete, p. 327
FC0B-FC2D,	Write a Word Marker Bit to Tape, p. 318
FC2E-FC32,	Write Data Bit to Tape, p. 320
FC33-FC3A,	Determine Which Part of Dipole Tape Write Routine Is Executing, p. 321
FC3B-FC49,	Prepare to Write Second Dipole for This Bit, p. 322
FC4A-FC53,	Prepare to Write Next Bit and Decrement Bit Counter, p. 323
FC54-FC6D,	Prepare Counters for Next Byte and Test If Writing Block Countdown Characters, p. 324
FC6E-FC7C,	Check for End of Tape Save, p. 325
FC7D-FC8B,	Move Next Byte from Save Area and Increment Pointer, p. 326
FC8C-FC94,	Prepare Parity Bit for This Byte, p. 326
FC95-FCA7,	Handle End-of-Block Processing and Reset IRQ Vector, p. 328
FCA8-FCCE,	Write Leader Bit to Tape and Reset IRQ Interrupt, p. 307
FCCF-FC5F,	Reset CIA/VIA Registers and Restore IRQ Vector, p. 333
FCF6-FD07,	Set IRQ Vector, p. 334
FD08-FD10,	Turn Off Tape Motor, p. 335
FD11-FD1A,	Compare Pointer to Current Byte with Pointer for End of Load/Save, p. 335
FD1B-FD21,	Increment Pointer to Current Byte, p. 336
FD22-FD3E,	System Reset, p. 17
FD3F-FD4C,	Test for Autostart Cartridge, p. 18
FD52-FD6C,	Initialize KERNAL Ram Vectors, p. 22
FD8D-FDFD,	Initialize Memory Pointers, p. 20
FDF9-FE48,	Initialize VIA Registers, p. 26
FE49-FE4F,	Set Filename Location And Number Of Characters, p. 105

## Appendix B

FE50–FE56,	Set Logical File Number, Device Number, Secondary Address, p. 103
FE57–FE6E,	Read/Set Status and Set Message Control, p. 97
FE6F–FE72,	Set Serial Timeout Value, p. 223
FE73–FE81,	MEMTOP Execution, p. 90
FE82–FE90,	MEMBOT Execution, p. 89
FE91–FEA8,	Test For RAM Byte, p. 21
FEA9–FEAC,	NMI Interrupt Handler Jump, p. 33
FEAD–FF5B,	NMI Interrupt Handler, p. 38
FED2–FEDD,	BRK Interrupt Handler, p. 47
FF72–FF84,	IRQ/BRK Interrupt Switch, p. 45
FF8A, RESTOR,	p. 98
FF8D, VECTOR,	p. 109
FF90, SETMSG,	p. 104
FF93, SECOND,	p. 102
FF96, TKSA,	p. 108
FF99, MEMTOP,	p. 89
FF9C, MEMBOT,	p. 89
FF9F, SCNKEY,	p. 101
FFA2, SETTMO,	p. 106
FFA5, ACPTR,	p. 68
FFA8, CIOUT,	p. 78
FFAB, UNTALK,	p. 109
FFAE, UNLSN,	p. 108
FFB1, LISTEN,	p. 85
FFB4, TALK,	p. 107
FFB7, READST,	p. 95
FFBA, SETLFS,	p. 102
FFBD, SETNAM,	p. 104
FFC0, OPEN,	p. 90
FFC3, CLOSE,	p. 79
FFC6, CHKIN,	p. 69
FFC9, CHKOUT,	p. 71
FFCC, CLRCHN,	p. 82
FFCF, CHRIN,	p. 73
FFD2, CHROUT,	p. 75
FFD5, LOAD,	p. 86
FFD8, SAVE,	p. 98
FFDB, SETTIN,	p. 105
FFDE, RDTIM,	p. 94
FFE1, STOP,	p. 106
FFE4, GETIN,	p. 83
FFE7, CLALL,	p. 78
FFEA, UDTIM,	p. 108
FFED, SCREEN,	p. 101
FFF0, PLOT,	p. 93
FFF3, IOBASE,	p. 84





## **Appendix C**

---

# **Cross-Reference of Kernal Routines by Chapter**



# Cross Reference of Kernal Routines by Chapter

---

## Chapter 2. System Reset

System Reset, FCE2/FD22-FD01/FD3E, p. 17  
Test for Autostart Cartridge, FD02/FD3F-FD0F/FD4C, p. 18  
Initialize Memory Pointers (64), FD50-FD9A, p. 19  
Initialize Memory Pointers (VIC), FD8D-FDFD, p. 20  
Test for RAM Byte (VIC), FE91-FEA8, p. 21  
Initialize Kernal RAM Vectors, FD15/FD52-FD2F/FD6C, p. 22  
Initialize I/O Chips (64), FDA3-FDF8 & FF6E-FF80, p. 24  
Initialize VIA Registers (VIC), FDF9-FE48, p. 26  
Initialize VIC-II Chip and Set PAL/NTSC Flag (64), FF5B-FF6D, p. 28

## Chapter 3. NMI Interrupts

NMI Interrupt Handler Jump, FE43/FEA9-FE46/FEAC, p. 33  
NMI Interrupt Handler (64), FE47-FEC1, p. 34  
NMI Interrupt Handler-Timer B Service (64), FED6-FF06, p. 37  
NMI Interrupt Handler-Start Timer B for FLAG NMI (64), FF07-FF2D, p. 37  
NMI Interrupt Handler (VIC), FEAD-FF5B, p. 38

## Chapter 4. IRQ Interrupts

IRQ/BRK Interrupt Switch, FF48/FF72-FF5A/FF84, p. 45  
BRK Interrupt Handler, FE66/FED2-FE71/FEDD, p. 47  
IRQ Interrupt Handler, EA31/EABF-EA86/EB1D, p. 48  
Jiffy Clock Update and STOP Key Scan, F69B/F734-F6DC/F75F, p. 51  
Keyboard Scan, EA87/EB1E-EB47/EBDB, p. 53  
Keyboard Table Setup, EB48/EBDC-EB78/EC45, p. 60

## Chapter 5. Kernal Routines

ACPTR, FFA5, p. 68  
CHKIN, FFC6, p. 69  
CHKIN Execution, F20E/F2C7-F236/F2EF, p. 70  
CHKOUT, FFC9, p. 71  
CHKOUT Execution, F250/F309-F278/F331, p. 72  
CHRIN, FFCE, p. 73  
Determine Input Device, F157/F20E-F178/F22F, p. 75  
CHROUT, FFD2, p. 75  
Determine Output Device, F1CA/F27A-F1E4/F28E, p. 76  
CINT (64), FF81, p. 77  
CIOUT, FFA8, p. 78  
CLALL, FFE7, p. 78

## Appendix C

Reset to No Open Files, F32F/F3EF-F332/F3F2, p. 78  
CLOSE, FFC3, p. 79  
Determine Device for CLOSE, F291/F34A-F2AA/F363, p. 80  
Common Exit For Close Logical File Routines, F2F1/F3B1-F30E/F3CE, p. 81  
CLRCHN, FFCC, p. 82  
Clear Serial Channels and Reset Default Devices, F333/F3F3-F349/F409,  
p. 82  
GETIN, FFE4, p. 83  
GETIN Preparation, F13E/F1F5-F14D/F204, p. 84  
IOBASE, FFF3, p. 84  
IONINIT (64), FF84, p. 85  
LISTEN, FFB1, p. 85  
LOAD, FFD5, p. 86  
Jump to LOAD Vector, F49E/F542-F4A4/F548, p. 88  
Determine Device for LOAD, F4A5/F549-F4B7/F55B &  
F533/F5CA-F538/F5D0, p. 88  
MEMBOT, FF9C, p. 89  
MEMBOT Execution, FE34/FE82-FE42/FE90, p. 89  
MEMTOP, FF99, p. 89  
MEMTOP Execution, FE25/FE73-FE33/FE81, p. 90  
OPEN, FFC0, p. 90  
OPEN Execution, F34A/F40A-F3D4/F494, p. 91  
PLOT, FFF0, p. 93  
RAMTAS (64), FF87, p. 93  
RDTIM, FFDE, p. 94  
RDTIM/SETTIM Execution, F6DD/F760-F6EC/F76F, p. 94  
READST, FFB7, p. 95  
Read/Set Status and Set Message Control, FE07/FE57-FE20/FE6E, p. 97  
RESTOR, FF8A, p. 98  
SAVE, FFD8, p. 98  
Jump to SAVE Vector, F5DD/F675-F5EC/F684, p. 100  
Determine Device for SAVE, F5ED/F685-F5F9/F691, p. 100  
SCNKEY, FF9F, p. 101  
SCREEN, FFED, p. 101  
SECOND, FF93, p. 102  
SETLFS, FFBA, p. 102  
Set Logical File Number, Device Number, Secondary Address,  
FE00/FE50-FE06/FE56, p. 103  
SETMSG, FF90, p. 104  
SETNAM, FFBD, p. 104  
Set Filename Location and Number of Characters, FDF9/FE49-FDFF/FE4F,  
p. 105,  
SETTIM, FFDB, p. 105,  
SETTMO, FFA2, p. 106,  
STOP, FFE1, p. 106,  
Test for STOP Key, F6ED/F770-F6FA/F77D, p. 107,  
TALK, FFB4, p. 107,  
TKSA, FF96, p. 108,  
UDTIM, FFEA, p. 108,  
UNLSN, FFAE, p. 108,

## Appendix C

UNTALK, FFAB, p. 109,  
VECTOR, FF8D, p. 109,

### Chapter 6. Miscellaneous Routines

Set I/O Defaults and Home Cursor, E59A/E5B5–E59F/E5BA, p. 113,  
Display Kernal Messages, F12B/F1E2–F13D/F1F4, p. 113  
See If Logical File Exists, F30F/F3CF–F31E/F3DE, p. 114  
Extract Logical File Number, Device Number, and Secondary Address from  
Tables, F31F/F3DF–F32E/F3EE, p. 115  
Display LOADING/VERIFYING Message, F5D2/F66A–F5DC/F674, p. 115  
Display SAVING Filename Message, F68F/F728–F69A/F733, p. 116  
Error Message Handler, F6FB/F77E–F72B/F7AE, p. 116  
Display SEARCHING FOR Message, F5AF/F647–F5C0/F658, p. 118  
Display Filename, F5C1/F659–F5D1/F669, p. 118

### Chapter 7. Screen Routines

Clear Screen Line Cursor Is On, E9FF/EA8D–EA12/EAA0, p. 126  
Set Pointers to Screen Line Cursor Is On, E9F0/EA7E–E9FE/EA8C, p. 127  
Set VIC Chip Registers, Clear Screen, Home Cursor, Set Screen Line Link  
Table, E518–E599/E5B4, p. 129  
Set Default Device Numbers, E5A0/E5BB–E5A7/E5C2, p. 133  
Initialize VIC Chip Registers, E5A8/E5C3–E5B3/E5CE, p. 134  
CHRIN from Keyboard or Screen, F157/F20E–F172/F229, p. 136  
Get Character from Keyboard or Screen, E632/E64F–E683/E6B7, p. 137  
Get Characters Until RETURN Key Detected, E5CA/E5E5–E631/E64E, p. 142  
If Quote Key Detected Then Flip Quote Flag, E684/E6B8–E690/E6C4, p. 145  
Set Color and Store Character on Screen, EA13/EAA1–EA1B/EAA9, p. 145  
Display Byte in Accumulator on Screen, EA1C/EAAA–EA23/EAB1, p. 145  
Set Pointer to Color Nybble, EA24/EAB2–EA30/EABE, p. 146  
Retrieve Character from Keyboard Queue, E5B4/E5CF–E5C9/E5E4, p. 147  
Main Screen Editor, E716/E742–E87B/E8C2, p. 148  
Display Screen Codes, E691/E6C5–E6A7/E6DB, p. 160  
Exit from Screen Editor Routines, E6A8/E6DC–E6B5/E6E9, p. 162  
Advance Cursor and Scroll or Insert Blank Lines, E6B6/E6EA–E700/E72C  
(VIC: also ED5B–ED68), p. 162  
Move Cursor to Previous Screen Line, E701/E72D–E75/E741, p. 165  
Advance Cursor To Next Screen Line, E87C/E8C3–E890/E8D7, p. 166  
Handle RETURN Key, E891/E8D8–E8A0/E8E7, p. 167  
Decrement Screen Line Pointer If Cursor Moves Left to New Line,  
E8A1/E8E8–E8B2/E8F9, p. 168  
Increment Screen Line Pointer If Cursor Moves Right to New Line,  
E8B3/E8FA–E8C1/E911, p. 169  
Test for Color Key, E8CB/E912–E8D9/E920, p. 170  
Scroll Screen, E8EA/E975–E964/E9ED, p. 171  
Insert Blank Line, E965/E9EE–E9C7/EA55, p. 174  
Move Screen Line, E9C8/EA56–E9DF/EA6D, p. 176  
Set Color Memory Pointers for Moving Line, E9E0/EA6E–E9EF/EA7D,  
p. 177  
Test for Character Set Switch, EC44/ED21–EC77/ED5A, p. 178  
Return Number of Columns and Rows in Screen, E505–E509, p. 179  
Read/Plot Cursor Location, E50A–E517, p. 179

## Appendix C

### Chapter 8. Serial I/O Routines

Bring Serial Bus Attention Line High, EDBE/EEC5–EDC6/EECD, p. 190  
Bring Serial Bus Data Line High, EE97/E4A0–EE9F/E4A8, p. 190  
Bring Serial Bus Data Line Low, EEA0/E4A9–EEA8/E4B1, p. 191  
Bring Serial Bus Clock Line High, EE85/EF84–EE8D/EF8C, p. 191  
Bring Serial Bus Clock Line Low, EE8E/EF8D–EE96/EF95, p. 192  
Read Serial Data In and Serial Clock In, EEA9/E4B2–EEB2/E4BB, p. 192  
Send LISTEN Command to Device, ED0C/EE17–ED10/EE1B, p. 193  
Do Attention Handshake with Serial Device, ED11/EE1C–ED3F/EE48, p. 194  
Send Serial Byte: Command or Data, ED40/EE49–EDAC/EEB3, p. 197  
Send OPEN, LOAD, or SAVE Command to Device, F3D5/F495–F408/F4C6, p. 202  
Send Secondary Address After LISTEN, EDB9/EEC0–EDBD/EEC4, p. 204  
Send Serial Byte Deferred, EDDD/EEE4–EDEE/EEF5, p. 205  
Set Status Word, EDAD/EEB4–EDB8/EEBF, p. 205  
Delay One Millisecond, EEB3/EF96–EEBA/EFA2, p. 206  
Save to Serial Device, F5FA/F692–F641/F6D9, p. 207  
Stop Load or Save, F633/F6CB–F639/F6D1, p. 209  
Send Secondary Address for CLOSE, F642/F6DA–F658/F6F0, p. 210  
Load or Verify from Serial Device, F4B8/F55C–F532/F5C9, p. 210  
Send TALK Command to Device, ED09/EE14–ED10/EE1B, p. 214  
Send Secondary Address After TALK and Do TALK-LISTEN Turnaround, EDC7/EECE–EDDC/EEE3, p. 214  
Receive Byte from Serial Device, EE13/EF19–EE84/EF83, p. 216  
Open Serial Input Channel, F237/F2F0–F24F/F308, p. 219  
Get Character from Serial Input Channel, F1AD/F264–F1B7/F26E, p. 220  
Open Serial Output Channel, F279/F332–F290/F349, p. 221  
Send UNTALK Command, EDEF/EEF6–EE12/EF18, p. 222  
Send UNLISTEN Command, EDFE/EF04–EE12/EF18, p. 222  
Close Logical File for Serial Device, F2EE/F3AE–F2F0/F3B0, p. 223  
Set Serial Timeout Value, FE21/FE6F–FE24/FE72, p. 223

### Chapter 9. RS-232 I/O Routines

Open Logical File for RS-232 Device, F409/F4C7–F482/F541, p. 233  
CIA Initialization for RS-232 (64), F483–F49D, p. 239  
Compute Bit Count, EF4A/F027–EF58/F035, p. 240  
Open RS-232 Channel for Input, F04D/F116–F085/F14E, p. 241  
CHRIN from RS-232 Device, F1B8/F26F–F1C9/F279, p. 243  
GETIN from RS-232 Device, F14E/F205–F156/F20D, p. 243  
Get Character from RS-232 Receive Buffer, F086/F14F–F0A3/F15F, p. 244  
Receive RS-232 Bit: NMI Interrupt Driven, EF59/F036–EF6D/F04A, p. 245  
Store Byte Received into Buffer, EF97/F06F–EFB2/F08A, p. 247  
Check Parity of Received Byte, EFB3/F08B–EFC6/F09E, p. 248  
Handle Errors While Receiving from RS-232 Device, EFC7/F09F–EFDA/F0B2, p. 250  
Check for Stop Bits, EF6E/F04B–EF7D/F05A, p. 250  
Check for Framing/Break Error, EFDB/F0B3–EFE0/F0B8, p. 252  
Prepare to Receive Next Byte, EF7E/F05B–EF8F/F067, p. 253  
Check for Start Bit, EF90/F068–EF96/F06E, p. 253  
Open RS-232 Channel for Output, EFE1/F0BC–F013/F0EC, p. 254

## Appendix C

CHROUT to RS-232 Device, F208/F2B9–F20D/F2C6, p. 255  
Store Character in Transmit Buffer, F014/F0ED–F02D/F101, p. 256  
Prepare Timer A/Timer 1 Interrupt for RS-232 Transmission, F02E/F102–F04C/F115, p. 257  
Prepare to Transmit Next Byte, EF06/EFEE–EF2D/F015, p. 258  
Handle Errors While Transmitting to RS-232 Device, EF2E/F016–EF49/F026, p. 259  
Transmit RS-232 Bit: NMI Interrupt Driven, EEBB/EFA3–EED6/EFBE, p. 260  
Prepare to Send Stop Bit, EF00/EFE8–EF05/EFED, p. 262  
Prepare Parity Bit and Set Counter for Stop Bits, EED7/EFBF–EEF1/EFEE, p. 263  
Close Logical File for RS-232 Device, F2AB/F364–F2C7/F38C, p. 264  
Disable RS-232 During Serial or Tape I/O, F0A4/F160–F0B6/F173, p. 266

### Chapter 10. Tape I/O Routines

Open Logical File for Writing to Tape, F3B8/F478–F3D4/F494, p. 292  
CHROUT to Tape, F1DC/F28F–F207/F2B8, p. 292  
Increment Count of Characters in Tape Buffer, F80D/F88A–F816/F893, p. 294  
Set Pointers to Start and End of Buffer and Write Buffer, F864/F8E3–F866/F8E5, p. 294  
Close Logical File for Tape, F2C8/F38D–F2ED/F3AD, p. 295  
Control Routine for Tape Save, F659/F6F1–F68E/F727, p. 296  
Load and Check Tape Buffer Address, F7D0/F84D–F7D6/F853, p. 299  
Set Start and End of Tape Buffer, F7D7/F854–F7E9/F866, p. 299  
Prepare Header and Write to Tape, F76A/F7E7–F7CF/F84C, p. 300  
Prepare to Write Program to Tape, F867/F8E6–F86A/F8E9, p. 302  
Prepare IRQ Vector and Timer Interrupts for Tape Write, F86B/F8EA–F874/F8F3, p. 302  
Reset IRQ Vector and Set Interrupt Enable Register, F875/F8F4–F8CF/F94A, p. 303  
Reverse Tape Write Line and Set Timer for Next Interrupt, FBA6/FBEA–FBC7/FC05, p. 305  
Write Leader Bit to Tape and Reset IRQ Interrupt, FC6A/FCA8–FC92/FCCE, p. 307  
Write a Word Marker Bit to Tape, FBCD/FC0B–FBF7/FC2D, p. 312  
Write Data Bit to Tape, FBF0/FC2E–FBF4/FC32, p. 320  
Determine Which Part of Dipole Tape Write Routine Is Executing, FBF5/FC33–FBF7/FC3A, p. 321  
Prepare to Write Second Dipole for this Bit, FBFD/FC3B–FC0B/FC49, p. 322  
Prepare to Write Next Bit and Decrement Bit Counter, FC0C/FC4A–FC15/FC53, p. 323  
Prepare Counters for Next Byte and Test if Writing Block Countdown Characters, FC16/FC54–FC2F/FC6D, p. 324  
Check for End of Tape Save, FC30/FC6E–FC3E/FC7C, p. 325  
Move Next Byte from Save Area and Increment Pointer, FC3F/FC7D–FC4D/FC8B, p. 326  
Prepare Parity Bit for this Byte, FC4E/FC8C–FC56/FC94, p. 326  
Indicate Block Save Complete, FBC8/FC06–FBCC/FC0A, p. 327  
Handle End-of-Block Processing and Reset IRQ Vector, FC57/FC95–FC69/FCA7, p. 328

## Appendix C

Check for Tape Button Down, F82E/F8AB-F837/F8B6, p. 329  
Display PRESS PLAY ON TAPE, F817/F894-F82D/F8AA, p. 330  
Display PRESS PLAY & RECORD ON TAPE, F838/F8B7-F840/F8BE, p. 331  
Check Keyboard Stop Key During Tape I/O, F8D0/F94B-F8E1/F95C, p. 331  
Reset Pointer to Start of Load/Save Area, FB8E/FBD2-FB96/FBDA, p. 332  
Reset Counters and Variables for Tape I/O, FB97/FBDB-FBA5/FBE9, p. 333  
Reset CIA/VIA Registers and Restore IRQ Vector, FC93/FCCF-FCB7/FC5F, p. 333  
Set IRQ Vector, FCB8/FCF6-FCC9/FD07, p. 334  
Turn Off Tape Motor, FCCA/FD08-FCD0/FD10, p. 335  
Compare Pointer to Current Byte with Pointer for End of Load/Save, FCD1/FD11-FCDA/FD1A, p. 335  
Increment Pointer to Current Byte, FCDB/FD1B-FCE1/FD21, p. 336  
Determine If Open Is for Read or Write, F38B/F44B-F398/F458, p. 337  
Open Logical File for Reading From Tape, F399/F459-F3D4/F494, p. 337  
CHRIN from Tape, F179/F230-F198/F24F, p. 339  
Return Byte from Tape Buffer, F199/F250-F1AC/F263, p. 340  
Control Routine for Tape Load, F539/F5D1-F5AE/F646, p. 341  
Find Specified Tape Header, F7EA/F867-F80C/F889, p. 346  
Find Next Tape Header, F72C/F7AF-F769/F7E6, p. 347  
Read Tape Header into Buffer, F841/F8C0-F849/F8C8, p. 349  
Load Next Two Blocks, F84A/F8C9-F863/F8E2, p. 349  
Determine Time Between FLAG/CA1 Interrupts for this Dipole, F92C/F98E-F939/F99B, p. 352  
Convert Time Between Interrupts into One-Byte Value, F93A/F99C-F95A/F9AF, p. 355  
Determine If Dipole Time Represents Noise, 0, 1, or Word Marker, F959/F9B0-F98A/F9E4, p. 358  
Set A8 If Bytes are Being Received, F98B/F9E5-F992/F9EC, p. 362  
Increment or Decrement the 0/1 Balanced Counter, F993/F9ED-F998/F9F2, p. 362  
Determine Value to Adjust Baseline Times, F999/F9F3-F9A1/F9FB, p. 363  
Flip Dipole Indicator Switch, F9A2/F9FC-F9A9/FA03, p. 366  
Store Dipole Value as Bit, F9AA/FA04-F9AB/FA05, p. 366  
Check Possible Error and See if Receiving Bytes, F9AC/FA06-F9AF/FA09, p. 367  
Determine if Interrupt Was Caused by Timer A/Timer 1 Timeout, F9B0/FA0A-F9C8/FA18, p. 367  
Determine if Parity for Byte Read is Correct, F9C9/FA19-F9D4/FA24, p. 368  
Set Adjustable Baseline Values for Next Bit, F9D5/FA25-F9E3/FA33, p. 370  
Determine If Two Dipoles Are Data, Error, or Leader Bit, F9E4/FA34-F9F6/FA46, p. 373  
Process Data or Parity Bit, F9F7/FA47-FA0F/FA5F, p. 375  
Process Word Marker Dipole, FA10/FA60-FA1E/FA6B, p. 376  
Word Marker Action, FA1F/FA6C-FA43/FA90, p. 377  
Determine if Dipole Is in Block or Leader, FA44/FA91-FA52/FA9F, p. 379  
Store Byte Received and Check Error Flags, FA53/FAA0-FA5F/FAAC, p. 380  
Determine Action to Take for this Byte, FA60/FAAD-FA85/FAD2 and FA8D/FADA-FA90/FADD, p. 381



## Appendix C

Check for Valid Block Countdown Characters, FA91/FADE–FAA4/FAF1 and FABA/FB07–FABF/FB0C, p. 382

Last Block Countdown Character, FAA5/FAF2–FAB9/FB0C, p. 384

Look for Initial Block Countdown Character, FA86/FAD3–FA89/FAD6, p. 385

Common Exit Point for RTI, FA8A/FAD7–FA8C/FAD9, p. 386

Valid Data Byte Received; Test for Short Block, FAC0/FB0D–FACD/FB1A, p. 386

Check for End of Load, FACE/FB1B–FAD5/FB22, p. 387

Determine Block Being Read, FAD6/FB23–FADA/FB27, p. 387

Load/Verify for Block One of Header or Program, FADB/FB28–FB07/FB54, p. 388

Block Two Processing, FB08/FB55–FB32/FB7F, p. 389

Flag Unrecoverable Read Error, FB33/FB80–FB39/FB86, p. 392

Load Byte and Increment Pointer to Load Area, FB3A/FB87–FB47/FB94, p. 392

End-of-Block Processing, FB48/FB95–FB67/FBAB, p. 393

Tape Load Completed, FB68/FBAC–FB8D/FBD1, p. 395

Set Timer A/Timer 1 Value to Tag Behind FLAG/CA1 Interrupt, F8E2/F95D–F92B/F98D, p. 396



# COMPUTE! Books

Ask your retailer for these **COMPUTE! Books** or order directly from **COMPUTE!**.

Call toll free (in US) **800-334-0868** (in NC 919-275-9809) or write COMPUTE! Books, P.O. Box 5058, Greensboro, NC 27403.

Quantity	Title	Price*	Total
_____	All About the Commodore 64, Volume 1	<b>\$12.95</b>	_____
_____	COMPUTE!'s First Book of Commodore 64	<b>\$12.95</b>	_____
_____	COMPUTE!'s Second Book of Commodore 64	<b>\$12.95</b>	_____
_____	COMPUTE!'s First Book of Commodore 64 Sound & Graphics	<b>\$12.95</b>	_____
_____	COMPUTE!'s Reference Guide to Commodore 64 Graphics	<b>\$12.95</b>	_____
_____	COMPUTE!'s Beginner's Guide to Commodore 64 Sound	<b>\$12.95</b>	_____
_____	COMPUTE!'s First Book of Commodore 64 Games	<b>\$12.95</b>	_____
_____	COMPUTE!'s Second Book of Commodore 64 Games	<b>\$12.95</b>	_____
_____	Commodore 64 Games for Kids	<b>\$12.95</b>	_____
_____	COMPUTE!'s Commodore Collection, Volume 1	<b>\$12.95</b>	_____
_____	Commodore Peripherals: A User's Guide	<b>\$ 9.95</b>	_____
_____	Creating Arcade Games on the Commodore 64	<b>\$14.95</b>	_____
_____	Machine Language Routines for the Commodore 64	<b>\$14.95</b>	_____
_____	Mapping the Commodore 64	<b>\$14.95</b>	_____
_____	The VIC and 64 Tool Kit: BASIC	<b>\$16.95</b>	_____
_____	Machine Language for Beginners	<b>\$14.95</b>	_____
_____	The Second Book of Machine Language	<b>\$14.95</b>	_____

\*Add \$2.00 per book for shipping and handling.  
Outside US add \$5.00 air mail or \$2.00 surface mail.

**Shipping & handling: \$2.00/book** \_\_\_\_\_  
**Total payment** \_\_\_\_\_

All orders must be prepaid (check, charge, or money order).

All payments must be in US funds.

NC residents add 4.5% sales tax.

☐ Payment enclosed.

Charge ☐ Visa ☐ MasterCard ☐ American Express

Acct. No. \_\_\_\_\_ Exp. Date \_\_\_\_\_

Name \_\_\_\_\_

Address \_\_\_\_\_

City \_\_\_\_\_ State \_\_\_\_\_ Zip \_\_\_\_\_

\*Allow 4-5 weeks for delivery.

Prices and availability subject to change.

Current catalog available upon request.



If you've enjoyed the articles in this book, you'll find the same style and quality in every monthly issue of **COM-PUTE!** Magazine. Use this form to order your subscription to **COMPUTE!**.

For Fastest Service  
Call Our **Toll-Free** US Order Line  
**800-334-0868**  
In NC call **919-275-9809**

## COMPUTE!

P.O. Box 5058  
Greensboro, NC 27403

My computer is:

- ☐ Commodore 64   ☐ TI-99/4A   ☐ Timex/Sinclair   ☐ VIC-20   ☐ PET  
☐ Radio Shack Color Computer   ☐ Apple   ☐ Atari   ☐ Other \_\_\_\_\_  
☐ Don't yet have one...

- ☐ \$24 One Year US Subscription  
☐ \$45 Two Year US Subscription  
☐ \$65 Three Year US Subscription

Subscription rates outside the US:

- ☐ \$30 Canada and Foreign Surface Mail  
☐ \$65 Foreign Air Delivery

Name \_\_\_\_\_

Address \_\_\_\_\_

City \_\_\_\_\_

State \_\_\_\_\_

Zip \_\_\_\_\_

Country \_\_\_\_\_

Payment must be in US funds drawn on a US bank, international money order, or charge card.

- ☐ Payment Enclosed   ☐ Visa  
☐ MasterCard   ☐ American Express

Acct. No. \_\_\_\_\_

Expires \_\_\_\_\_

/ (Required)

Your subscription will begin with the next available issue. Please allow 4-6 weeks for delivery of first issue. Subscription prices subject to change at any time.



If you've enjoyed the articles in this book, you'll find the same style and quality in every monthly issue of **COMPUTE!'s Gazette** for Commodore.

For Fastest Service  
Call Our **Toll-Free** US Order Line  
**800-334-0868**  
In NC call **919-275-9809**

## COMPUTE!'s Gazette

P.O. Box 5058  
Greensboro, NC 27403

My computer is:

☐ Commodore 64    ☐ VIC-20    ☐ Other \_\_\_\_\_

- ☐ \$24 One Year US Subscription  
☐ \$45 Two Year US Subscription  
☐ \$65 Three Year US Subscription

Subscription rates outside the US:

- ☐ \$30 Canada  
☐ \$65 Air Mail Delivery  
☐ \$30 International Surface Mail

Name \_\_\_\_\_

Address \_\_\_\_\_

City \_\_\_\_\_

State \_\_\_\_\_

Zip \_\_\_\_\_

Country \_\_\_\_\_

Payment must be in US funds drawn on a US bank, international money order, or charge card. Your subscription will begin with the next available issue. Please allow 4-6 weeks for delivery of first issue. Subscription prices subject to change at any time.

- ☐ Payment Enclosed    ☐ Visa  
☐ MasterCard    ☐ American Express

Acct. No. \_\_\_\_\_

Expires \_\_\_\_\_

/

(Required)

The *COMPUTE!'s Gazette* subscriber list is made available to carefully screened organizations with a product or service which may be of interest to our readers. If you prefer not to receive such mailings, please check this box ☐.





# COMPUTE! Books

Ask your retailer for these **COMPUTE! Books** or order directly from **COMPUTE!**.

Call toll free (in US) **800-334-0868** (in NC 919-275-9809) or write COMPUTE! Books, P.O. Box 5058, Greensboro, NC 27403.

Quantity	Title	Price*	Total
_____	Machine Language for Beginners (11-6)	<b>\$14.95</b>	_____
_____	The Second Book of Machine Language (53-1)	<b>\$14.95</b>	_____
_____	COMPUTE!'s Guide to Adventure Games (67-1)	<b>\$12.95</b>	_____
_____	Computing Together: A Parents & Teachers Guide to Computing with Young Children (51-5)	<b>\$12.95</b>	_____
_____	Personal Telecomputing (47-7)	<b>\$12.95</b>	_____
_____	BASIC Programs for Small Computers (38-8)	<b>\$12.95</b>	_____
_____	Programmer's Reference Guide to the Color Computer (19-1)	<b>\$12.95</b>	_____
_____	Home Energy Applications (10-8)	<b>\$14.95</b>	_____
_____	The Home Computer Wars: An Insider's Account of Commodore and Jack Tramiel		
_____	Hardback (75-2)	<b>\$16.95</b>	_____
_____	Paperback (78-7)	<b>\$ 9.95</b>	_____
_____	The Book of BASIC (61-2)	<b>\$12.95</b>	_____
_____	Every Kid's First Book of Robots and Computers (05-1)	<b>\$ 4.95†</b>	_____
_____	The Beginner's Guide to Buying a Personal Computer (22-1)	<b>\$ 3.95†</b>	_____
_____	The Greatest Games: The 93 Best Computer Games of all Time (95-7)	<b>\$ 9.95</b>	_____

\* Add \$2.00 per book for shipping and handling.  
† Add \$1.00 per book for shipping and handling.  
Outside US add \$5.00 air mail or \$2.00 surface mail.

**Shipping & handling: \$2.00/book** \_\_\_\_\_  
**Total payment** \_\_\_\_\_

All orders must be prepaid (check, charge, or money order).

All payments must be in US funds.

NC residents add 4.5% sales tax.

☐ Payment enclosed.

Charge ☐ Visa ☐ MasterCard ☐ American Express

Acct. No. \_\_\_\_\_ Exp. Date \_\_\_\_\_

Name \_\_\_\_\_ (Required)

Address \_\_\_\_\_

City \_\_\_\_\_ State \_\_\_\_\_ Zip \_\_\_\_\_

\*Allow 4-5 weeks for delivery.

Prices and availability subject to change.

Current catalog available upon request.









# Computing with the Kernal

When you turn on your Commodore 64 or VIC computer and the screen displays the familiar BYTES FREE message, you're seeing the Kernal is at work. In fact, you probably seldom use the VIC or 64 without accessing the Kernal. The Kernal routines (named for a misspelled version of the word *kernel*, meaning essence or core) make up the operating system of the Commodore home computers.

*COMPUTE!'s VIC-20 and Commodore 64 Tool Kit: The Kernal* is your guide to the heart of your computer. Thoroughly documented and clearly written, it describes the mysterious inner workings of the Kernal Read Only Memory (ROM).

Learning about the Kernal is important to anyone wanting to fully understand Commodore computers. Understanding the Kernal is particularly important if you're programming machine language routines—either for use with BASIC programs or as stand-alone programs. If you already know machine language, *Tool Kit: The Kernal* can help you determine the best place to jump into the ROM routines to keep your programs as short as possible. It will also help you trap the bugs in programs you've already written by allowing you to follow the process step-by-step. The Kernal descriptions in this book include:

- System Reset: initializing the computer
- IRQ Interrupts: using IRQ interrupts in your program
- NMI Interrupts: understanding hardware interrupts
- The Kernal Jump Table: wedging into the Kernal
- Screen Routines: displaying information on the screen
- Serial Input/Output: sending and receiving serial information
- RS-232 Input/Output: handling RS-232 communications
- Tape Input/Output: writing to and reading from cassette tape

*COMPUTE!'s VIC-20 and Commodore 64 Tool Kit: The Kernal* is a significant resource for all programmers. If you want to understand and use the machine language routines already in your computer, this reference guide will be a vital part of your programming library.